新工科·普通高校程序设计优秀教材



Java编程指南

语法基础、面向对象、函数式编程与项目实战

关东升 著

清华大学出版社

Java编程指南

——语法基础、面向对象、函数式编程与项目实战

关东升 著

清华大学出版社
北 京

内 容 简 介

本书是一部系统论述 Java 编程语言的教程，主要内容包括绪论、开发环境搭建、第一个 Java 程序、Java 语法基础、数据类型、运算符、控制语句、数组、字符串、面向对象基础、对象、继承与多态、抽象类与接口、Java 常用类、内部类、Lambda 表达式、异常处理、集合、泛型、文件管理与 I/O 流、多线程编程、网络编程、Swing 图形用户界面编程、数据库编程。本书最后给出了一个实战项目——开发 PetStore 宠物商店项目。每一章后面都安排若干道同步练习题，并在附录 C 中提供了参考答案。

本书可作为高等学校计算机软件技术课程的教材，也可作为社会培训机构的培训教材，还可作为广大 Java 初学者和 Java 开发的程序员的参考书。

图书在版编目（CIP）数据

Java 编程指南：语法基础、面向对象、函数式编程与项目实战/关东升著.—北京：清华大学出版社，2019

ISBN 978-7-302-52683-4

Ⅰ. ①J…　Ⅱ. ①关…　Ⅲ. ①JAVA 语言－程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2019)第 057432 号

责任编辑：盛东亮
封面设计：李召霞
责任校对：李建庄
责任印制：宋　林

出版发行：清华大学出版社
　　网　　址：http://www.tup.com.cn，http://www.wqbook.com
　　地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
　　社 总 机：010-62770175　　**邮　　购**：010-62786544
　　投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
　　质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
　　课件下载：http://www.tup.com.cn，010-62795954
印 装 者：三河市铭诚印务有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：28.5　　**字　　数**：691 千字
版　　次：2019 年 8 月第 1 版　　**印　　次**：2019 年 8 月第1 次印刷
定　　价：79.00 元

产品编号：082636-01

前 言

PREFACE

Java 语言经过 20 多年不断发展，变得更加成熟、更加易用。多年来 Java 语言一直是非常受欢迎的语言，这也说明了 Java 语言的生命力。Java 语言最开始用于设计开发机顶盒，经过发展，目前主要用于 Java Web 应用、企业级应用、Android 应用和桌面应用开发。

书中源代码

1. 源代码下载

书中包括了 200 多个完整示例，以及一个完整的案例项目源代码，读者可以到本书网站 http://www.zhijieketang.com/group/9 下载。

2. 源代码目录结构

作为一本介绍编程的书，本书提供很多示例源代码。下载本书源代码并解压，会看到如图 0-1 所示的目录结构。图 0-1 中的 ch5.6 表示第 5.6 节的示例代码，A.3.2 表示附录 A 中 3.2 节，B 表示附录 B 中的代码，每个文件夹都是一个 Eclipse 项目。

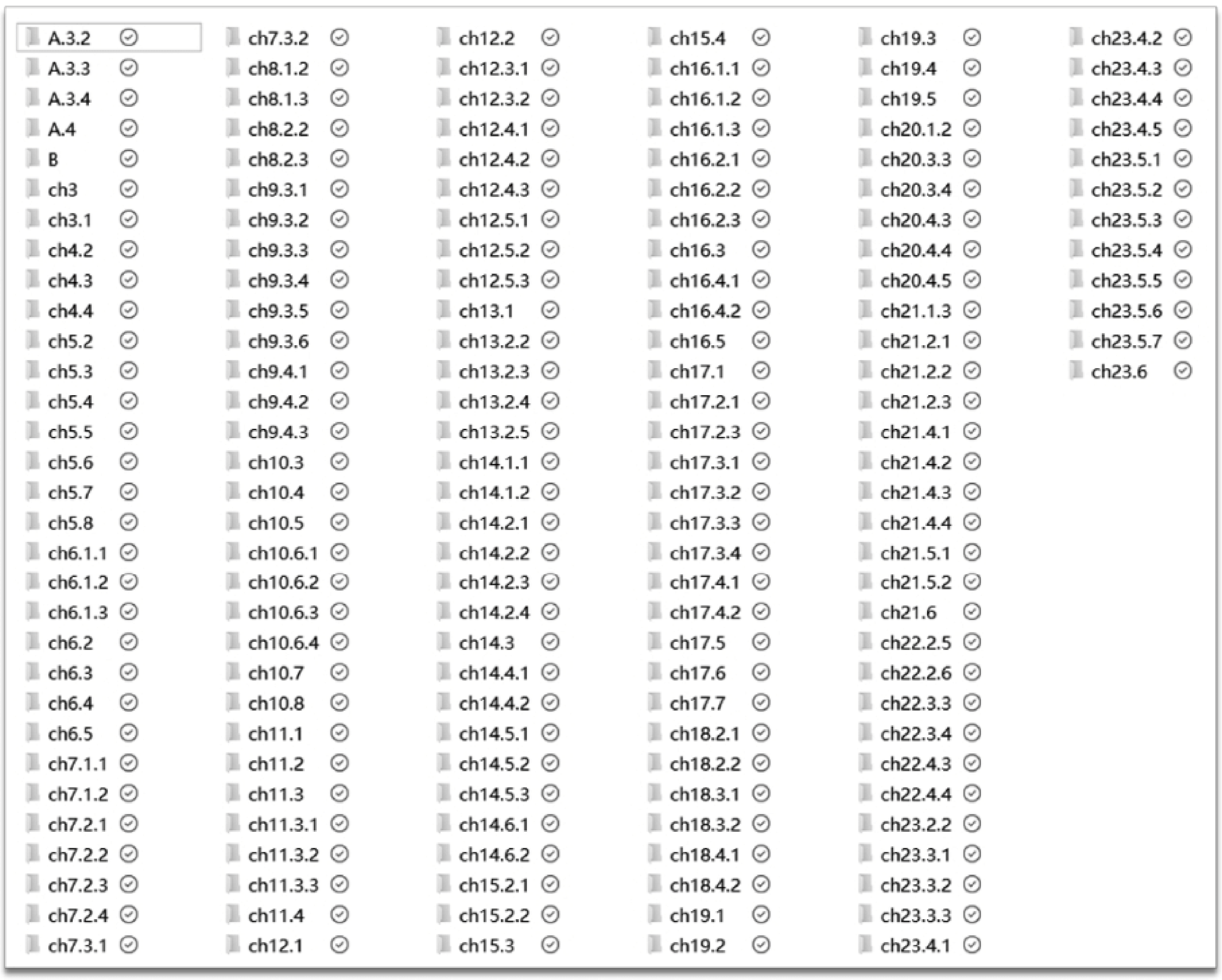


图 0-1　示例源代码目录结构

3. 导入 Eclipse 源代码项目

如何将 Eclipse 源代码项目导入到自己的 Eclipse 中呢？可以在 Eclipse 工具中选择“文件”→“导入”命令，打开如图 0-2 所示的导入项目对话框，在对话框中选中 General 下的“现有项目到工作空间中”，然后单击“下一步”按钮进入下一个对话框。如图 0-3 所示，单击“选择根目录”后面的“浏览”按钮选择源代码目录，这样会找到该目录下所有的 Eclipse 项目，根据自己的需要选中项目，然后单击“完成”按钮，将项目导入到 Eclipse 中。



图 0-2 导入项目对话框(1)



图 0-3 导入项目对话框(2)

另外，笔者推荐在导入时将项目复制到自己的工作空间中，这需要选中如图 0-3 所示“选项”选项区域中的“将项目复制到工作空间中”复选框。

勘误与支持

我们在网站 http://www.zhijieketang.com/group/9 中建立了一个勘误专区，可以及时地把书中的问题、失误和纠正反馈给广大读者。如果读者发现了任何问题，均可以在网上留言，也可以发送电子邮件到 eorient@sina.com，我们会在第一时间给予回复。

致谢

在此，感谢清华大学出版社的盛东亮编辑给我们提供了宝贵的意见。本书主要由关东升编写。此外，赵志荣、赵大羽、关锦华、闫婷娇、关秀华、王馨然、闫喜华和赵浩丞参与了部分内容的写作。感谢赵浩丞手绘了书中全部草图，并从专业的角度修改书中图片，力求更加真实完美地奉献给广大读者。

由于时间仓促，书中难免存在不妥之处，请读者见谅，并提出宝贵意见。

关东升

2019 年 4 月

目录

CONTENTS

第 1 章 CHAPTER 1 绪　　论

Java 诞生到现在已经有 20 多年了，但是 Java 仍然是非常热门的编程语言之一，很多平台使用 Java 语言开发。表 1-1 所示的是 TIOBE 社区发布的 2017 年 5 月和 2018 年 5 月的编程语言排行榜，可见 Java 语言的热度，或许这也是很多人选择学习 Java 的主要原因。

表 1-1　TIOBE 编程语言排行榜

2018 年 5 月	2017 年 5 月	变　化	编 程 语 言	评　　级	评 级 变 化
1	1		Java	14.639%	−6.320%
2	2		C	7.002%	−6.220%
3	3		C++	4.751%	−1.950%
4	5	↑	Python	3.548%	−0.240%
5	4	↓	C#	3.457%	−1.020%
6	10	⇈	Visual Basic .NET	3.391%	1.070%
7	7		JavaScript	3.071%	0.730%
8	12	⇈	Assembly Language	2.859%	0.980%
9	6	↓	PHP	2.693%	−0.300%
10	9	↓	Perl	2.602%	0.280%
11	8	↓	Ruby	2.429%	0.090%
12	13	↑	Visual Basic	2.347%	0.520%
13	15	↑	Swift	2.274%	0.680%
14	16	↑	R	2.192%	0.860%
15	14	↓	Objective-C	2.101%	0.500%
16	42	⇈	Go	2.080%	1.830%
17	18	↑	MATLAB	2.063%	0.780%
18	11	⇊	Delphi/Object Pascal	2.038%	0.030%
19	19		PL/SQL	1.676%	0.470%
20	22	↑	Scratch	1.668%	0.740%

1.1　Java 语言的发展历史

在正式学习 Java 语言之前，读者有必要先来了解一下 Java 的历史。1990 年底，美国 Sun 公司[1]成立了一个叫作 Green 的项目组，该 Green 项目主要目标是为消费类电子产品开发一种分布式系统，使之能够操控电冰箱、电视机等家用电器。

① Sun 公司创建于 1982 年，主要产品是工作站及服务器。1986 年在美国成功上市，1992 年 Sun 推出了市场上第一台多 CPU 台式机，1993 年进入财富 500 强，1995 年开发了 Java 语言，2010 年被 Oracle（甲骨文）公司收购。现在 Java 技术是由 Oracle 公司提供的。

消费类电子产品种类很多，包括掌上电脑（个人数字助理，Personal Digital Assistant，PDA）、机顶盒、手机等，这些消费类电子产品所采用的处理芯片和操作系统基本上都是不相同的，存在跨平台等问题。开始时，Green 项目组考虑采用 C++语言来编写消费类电子产品的应用程序，但是 C++语言过于复杂、庞大，而且安全性差。于是他们设计并开发出一种新的语言——Oak（橡树）。Oak 这个名字来源于 Green 项目组办公室窗外的一棵橡树。由于 Oak 在进行注册商标时已经被注册，他们需要为这个新语言取一个新的名字。有一天，几位项目的成员正在咖啡馆喝着 Java（爪哇）咖啡，其中一个人灵机一动说就叫 Java 怎么样？马上得到了其他人的同意，于是这个新的语言取名为 Java。

Sun 公司在 1996 年发布了 Java 1.0，但是 Java 1.0 开发的应用速度很慢，并不适合做真正的应用开发，直到 Java 1.1 速度才有了明显的提升。Java 设计之初是为消费类电子产品开发应用，但是真正使 Java 流行起来是在互联网上的 Web 应用程序。20 世纪 90 年代正处于互联网发展起步阶段，互联网上设备差别很大，需要应用程序能够跨平台运行，Java 语言就具有“一经编写到处运行”的跨平台能力。

到本书编写时，Oracle 公司已经发布了 Java 10。Java 在 20 多年的发展过程中，与时俱进，为了适应时代的需要，经历过两次重大的版本升级：一个是 Java 5，Java 5 提供了泛型等重要的功能；另一个是 Java 8，Java 8 提供了 Lambda 表达式和枚举类等重要的功能。

1.2 Java 语言的特点

Java 语言能够流行起来，并长久不衰，得益于 Java 语言有很多优秀的关键特点。这些特点包括简单、面向对象、分布式、结构中立、可移植、解释执行、健壮、安全、高性能、多线程和动态。下面给出详细解释。

1. 简单

Java 设计目标之一就是能够方便学习，使用简单。由于当初 C++程序员很多，介绍 C++语言的书籍也很多，所以 Java 语言的风格设计成为类似于 C++语言风格，但 Java 摒弃了 C++中容易引发程序错误的地方，如指针、内存管理、运算符重载和多继承等。一方面 C++程序员可以很快迁移到 Java；另一方面没有编程经验的初学者也能很快学会 Java。

2. 面向对象

面向对象是 Java 最重要的特性。Java 是彻底的、纯粹的面向对象语言，在 Java 中“一切都是对象”。Java 完全具有面向对象的三个基本特性：封装、继承和多态，其中封装性实现了模块化和信息隐藏，继承性实现了代码的复用，用户可以建立自己的类库。而且 Java 采用的是相对简单的面向对象技术，去掉了多继承等复杂的概念，只支持单继承。

3. 分布式

Java 语言就是为分布式系统而设计的。JDK（Java Development Kits，Java 开发工具包）中包含了支持 HTTP 和 FTP 等基于 TCP/IP 协议的类库。Java 程序可以凭借 URL 打开并访问网络上的对象，其访问方式与访问本地文件系统几乎完全相同。

4. 结构中立

Java 程序需要在很多不同网络设备中运行，这些设备有很多不同类型的计算机和操作

系统。为能够使 Java 程序在网络的任何地方运行，Java 编译器编译生成了与机器结构(CPU 和操作系统)无关的字节码(byte-code)文件。任何类型的计算机，只要可以运行 Java 虚拟机，字节码文件就可以在该计算机上运行。

5. 可移植

体系结构的中立也使得 Java 程序具有可移植性。针对不同的 CPU 和操作系统，Java 虚拟机有不同的版本，这样就可以保证相同的 Java 字节码文件可以移植到多个不同的平台上运行。

6. 解释执行

为实现跨平台，Java 设计成为解释执行的，即 Java 源代码文件首先被编译成为字节码文件，这些字节码本身包含了许多编译时生成的信息，在运行时 Java 解释器负责将字节码文件解释成为特定的机器码进行运行。

7. 健壮

Java 语言是强类型语言，它在编译时进行代码检查，使得很多错误能够在编译期被发现，不至于在运行期发生而导致系统崩溃。

Java 摒弃了 C++中指针操作，指针是一种强大的技术，能够直接访问内存单元，但同时也很复杂，如果指针操控不好，会引起导致内存分配错误、内存泄漏等问题。而 Java 中则不会出现由指针所导致的问题。

内存管理方面，C/C++等语言采用手动分配和释放，经常会导致内存泄漏，从而导致系统崩溃。而 Java 采用自动内存垃圾回收机制①，程序员不再需要管理内存，从而减少内存错误的发生，提高了程序的健壮性。

8. 安全

在 Java 程序执行过程中，类装载器负责将字节码文件加载到 Java 虚拟机中，这个过程中由字节码校验器检查代码中是否存在着非法操作。如果字节码校验器检验通过，由 Java 解释器负责把该字节码解释成为机器码进行执行，这种检查可以防止木马病毒。

另外，Java 虚拟机采用的是"沙箱"运行模式，即把 Java 程序的代码和数据都限制在一定内存空间里执行，不允许程序访问该内存空间外的内存。

9. 高性能

Java 编译器在编译时对字节码会进行一些优化，使之生成高质量的代码。Java 字节码格式就是针对机器码转换而设计的，实际转换时相当简便。Java 在解释运行时采用一种即时编译技术，可使 Java 程序的执行速度提升很大。经过多年的发展，Java 虚拟机也有很多改进，这都使得 Java 程序的执行速度提升很大。

10. 多线程

Java 是为网络编程而设计的，这要求 Java 能够并发处理多个任务。Java 支持多线程编程，多线程机制可以实现并发处理多个任务，互不干涉，不会由于某一任务处于等待状态而影响了其他任务的执行，这样就可以容易地实现网络上的实时交互操作。

---

① 在 Java 运行环境中，始终存在着一个系统级的线程，专门跟踪内存的使用情况，定期检测出不再使用的内存，并进行自动回收，避免了内存的泄漏，也减轻了程序员的工作量。

11. 动态

Java 应用程序在运行过程中可以动态地加载各种类库,即使更新类库也不必重新编译使用这一类库的应用程序。这一特点使之非常适合于在网络环境下运行,同时也非常有利于软件的开发。

1.3 Java 平台

Java 不仅是编程语言,而且是一个开发平台,Sun 公司根据 Java 应用领域的不同将 Java 分成三个平台:Java SE、Java EE 和 Java ME。

1.3.1 Java SE

Java SE 是 Java Standard Edition 的简写,主要用于为台式机和工作站的桌面应用(application)程序。Java SE 是其他平台的基础,本书主要介绍 Java SE 版本中的技术。

Java SE 中主要包含了 JRE(Java SE Runtime Environment,Java SE 运行环境)、JDK(Java Development Kit,Java 开发工具包)和 Java 核心类库。如果只是运行 Java 程序,不考虑开发 Java 程序,那么只安装 JRE 就可以了。在 JRE 中包含了 Java 程序运行所需要的 Java 虚拟机(Java Virtual Machine,JVM)。JDK 中包含了 JRE 和一些开发工具,这些工具包括编译器、文档生成器和文件打包等工具。

另外,Java SE 中还提供了 Java 应用程序开发需要的基本的和核心的类库,这些类库包括字符串、集合、输入/输出、网络通信和图形用户界面等。事实上,学习 Java 就是在学习 Java 语法和 Java 类库的使用。

1.3.2 Java EE

Java EE 是 Java Enterprise Edition 的简写,主要目的是为简化企业级系统的开发、部署和管理。Java EE 是以 Java SE 为基础的,并提供了一套服务、API 接口和协议,能够开发企业级分布式系统、Web 应用程序和业务组件等,其中包括 JSP、Servlet、EJB、JNI 和 Java Mail 等。

1.3.3 Java ME

Java ME 是 Java Micro Edition 的简写,主要是面向消费类电子产品,为消费电子产品提供一个 Java 的运行平台,使得 Java 程序能够在手机、机顶盒、PDA 等产品上运行。Java ME 在早期的诺基亚塞班手机系统有很多应用,而现在的 iOS 和 Android 等智能手机中基本上没有它的用武之地。

1.4 Java 虚拟机

Java 应用程序能够跨平台运行,主要是通过 Java 虚拟机实现的。如图 1-1 所示,不同软硬件平台 Java 虚拟机是不同的,Java 虚拟机往下是不同的操作系统和 CPU,使用或开发时需要下载不同的 JRE 或 JDK 版本。Java 虚拟机往上是 Java 应用程序,Java 虚拟机屏蔽

了不同软硬件平台，Java 应用程序不需要修改，不需要重新编译，直接可以在其他平台上运行。

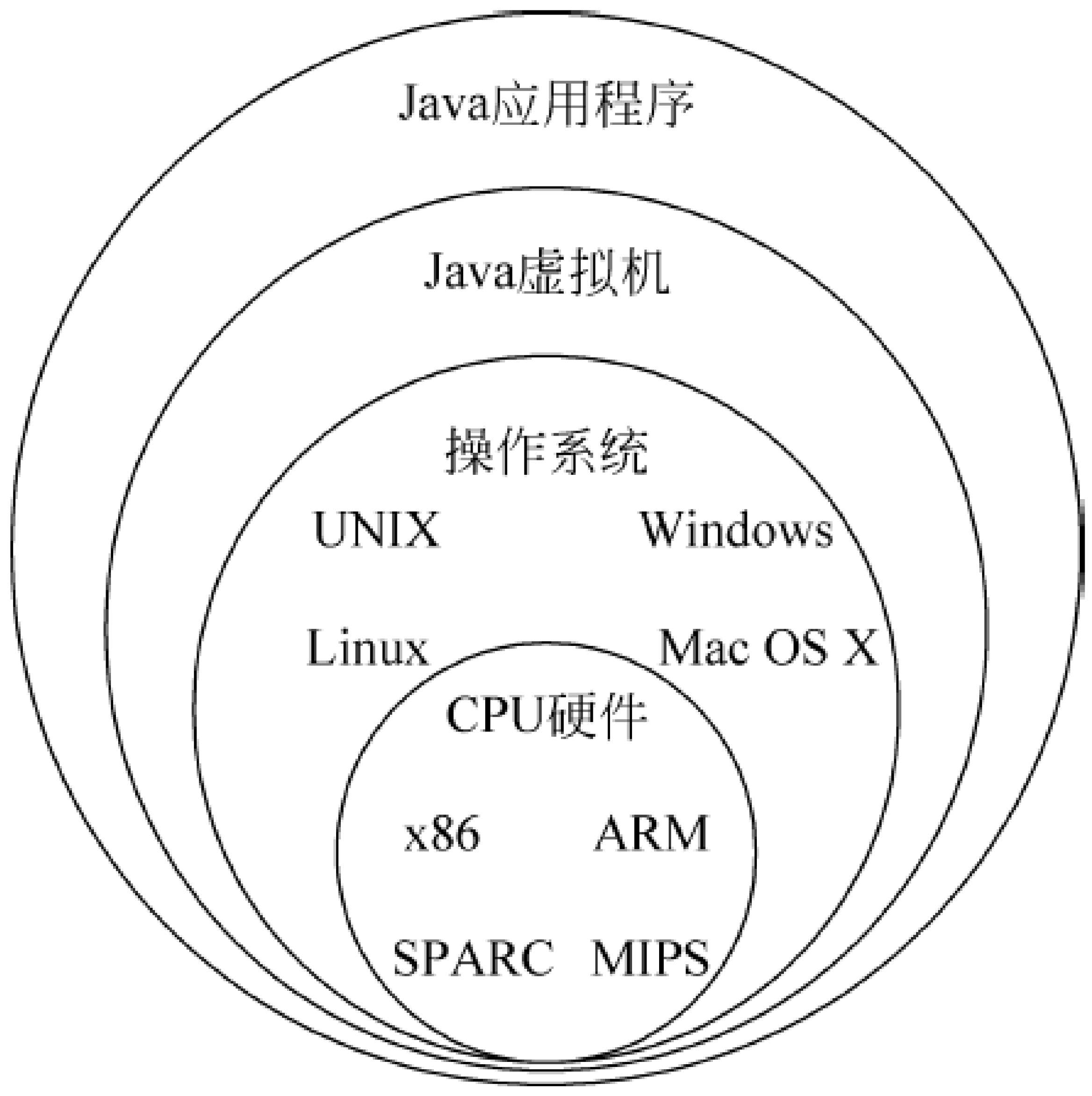


图 1-1 Java 虚拟机

Java 虚拟机中包含了 Java 解释器，Java 程序运行过程如图 1-2 所示，首先由编译器将 Java 源程序文件(.java 文件)编译成为字节码文件(.class 文件)，然后再由 Java 虚拟机中的解释器将字节码解释成机器码去执行。

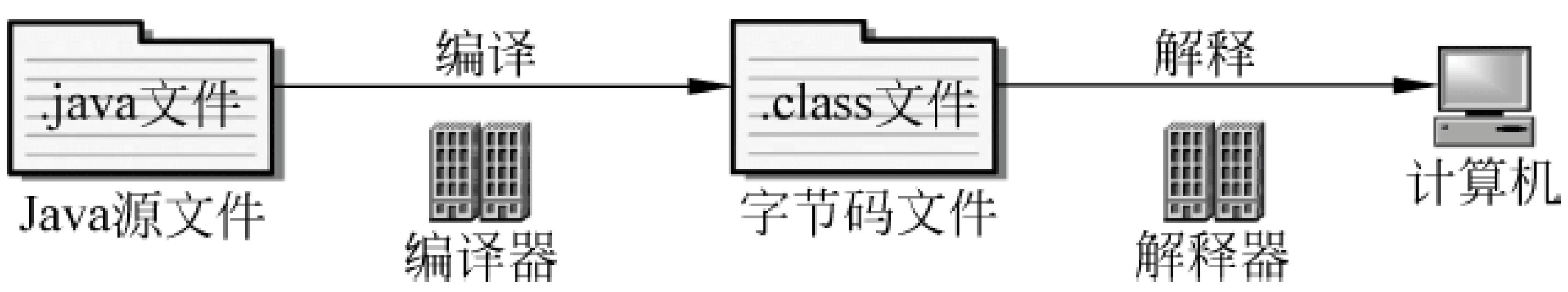


图 1-2 Java 程序运行过程

本章小结

本章首先介绍了 Java 的发展历史、Java 语言的特点，然后介绍了 Java 三大平台，最后介绍了 Java 虚拟机。

1.5 同步练习

1. Java 语言的特点有哪些?
2. Java 平台有哪些?

第 2 章 开发环境搭建

CHAPTER 2

《论语·魏灵公》曰："工欲善其事，必先利其器"，做好一件事，准备工作非常重要。在开始学习 Java 技术之前，先介绍如何搭建 Java 开发环境是非常重要的一件事情。

Oracle 公司提供的 JDK 只是一个开发工具包，它不是一个 IDE（Integrated Development Environments，集成开发环境），IDE 的开发工具将程序的编辑、编译、调试、执行等功能集成在一个开发环境中，使用户可以很方便地进行软件的开发。Java 开发 IDE 工具有很多，其中主要有 Eclipse、IntelliJ IDEA 和 NetBeans 等。

2.1 JDK 工具包

JDK 工具包是最基础的 Java 开发工具，很多 Java IDE 工具，如 Eclipse、IntelliJ IDEA 和 NetBeans 等都依赖于 JDK，也有一些人使用"JDK＋文本编辑工具"编写 Java 程序。

2.1.1 JDK 下载和安装

截止本书编写完成为止，Oracle 公司对外发布的是最新 JDK 8。图 2-1 所示是 JDK 8 下载界面，它的下载地址是 http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html。其中有很多版本，支持的操作系统有 Linux、Mac OS X[①]、Solaris[②] 和 Windows。注意选择对应的操作系统，以及 32 位还是 64 位安装的文件。

如果计算机是 Windows 10 64 位系统，则首先选中 Accept License Agreement（同意许可协议）单选按钮，然后单击 jdk-8u131-windows-x64.exe 下载 JDK 文件。

下载完成后，双击 jdk-8u131-windows-x64.exe 文件就可以安装了，安装过程中会弹出如图 2-2 所示的内容选择对话框，其中"开发工具"是 JDK 内容；"源代码"是安装 Java SE 源代码文件，如果安装源代码，安装完成后会弹出如图 2-3 所示的 src.zip 文件，这就是源代码文件；公共 JRE 就是 Java 运行环境，这里可以不安装，因为 JDK 文件夹中也会有一个 JRE，如图 2-3 所示的 jre 文件夹。

2.1.2 设置环境变量

安装完成之后，需要设置环境变量，主要包括：

---

① 苹果桌面操作系统，基于 UNIX 操作系统，现在改名为 macOS。

② 原 Sun 公司 UNIX 操作系统，现在被 Oracle 公司收购。



图 2-1　下载 JDK 8 界面



图 2-2　安装内容选择对话框

（1）JAVA_HOME 环境变量，指向 JDK 目录，很多 Java 工具运行都需要 JAVA_HOME 环境变量，所以推荐添加该变量。

（2）将 JDK\bin 目录添加到 Path 环境变量中，这样在任何路径下都可以执行 JDK 提供的工具指令。

首先需要打开 Windows 系统环境变量设置对话框，打开该对话框有很多方式，如果是

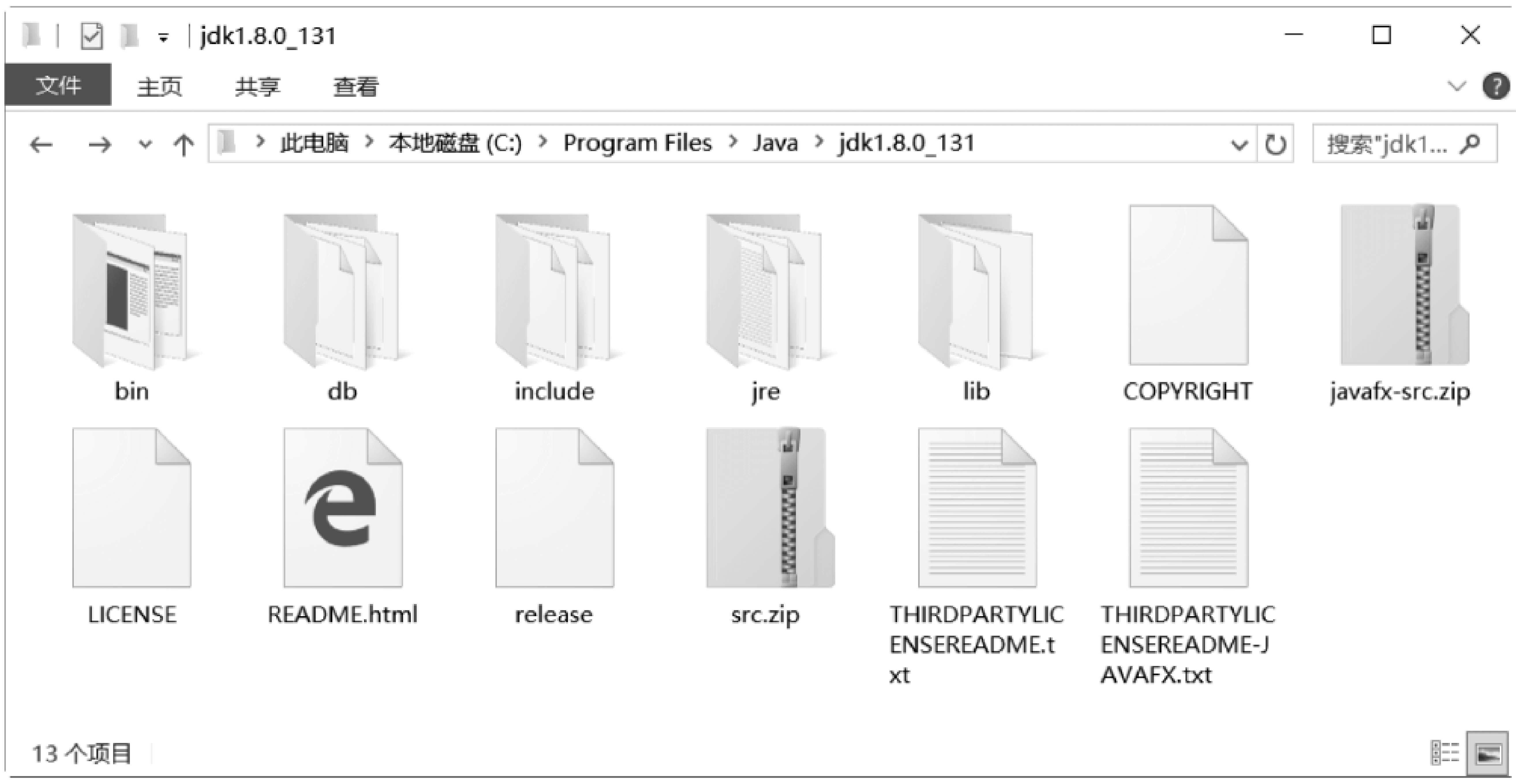


图 2-3 JDK 安装后的内容

Windows 10 系统，则打开步骤是：右击屏幕左下角的 Windows 图标，单击“系统”菜单，然后弹出如图 2-4 所示的 Windows 系统对话框，单击左边的“高级系统设置”超链接，打开如图 2-5 所示的高级系统设置对话框。



图 2-4 Windows 系统对话框



图 2-5　高级系统设置对话框

在如图 2-5 所示的高级系统设置对话框中，单击“环境变量”按钮打开环境变量设置对话框，如图 2-6 所示，可以在用户变量（上半部分，只配置当前用户）或系统变量（下半部分，配置所有用户）添加环境变量。一般情况下，在用户变量中设置环境变量。



图 2-6　环境变量设置对话框

在用户变量部分单击“新建”按钮，系统弹出对话框，如图 2-7 所示。将“变量名”设置为 JAVA_HOME，“变量值”设置为 JDK 安装路径。最后单击“确定”按钮完成设置。然后追加 Path 环境变量，在用户变量中找到 Path，双击 Path，弹出 Path 变量对话框，如图 2-8 所示，追加%JAVA_HOME%\bin。注意，多个变量路径之间用分号(;)分隔。最后单击“确定”按钮完成设置。

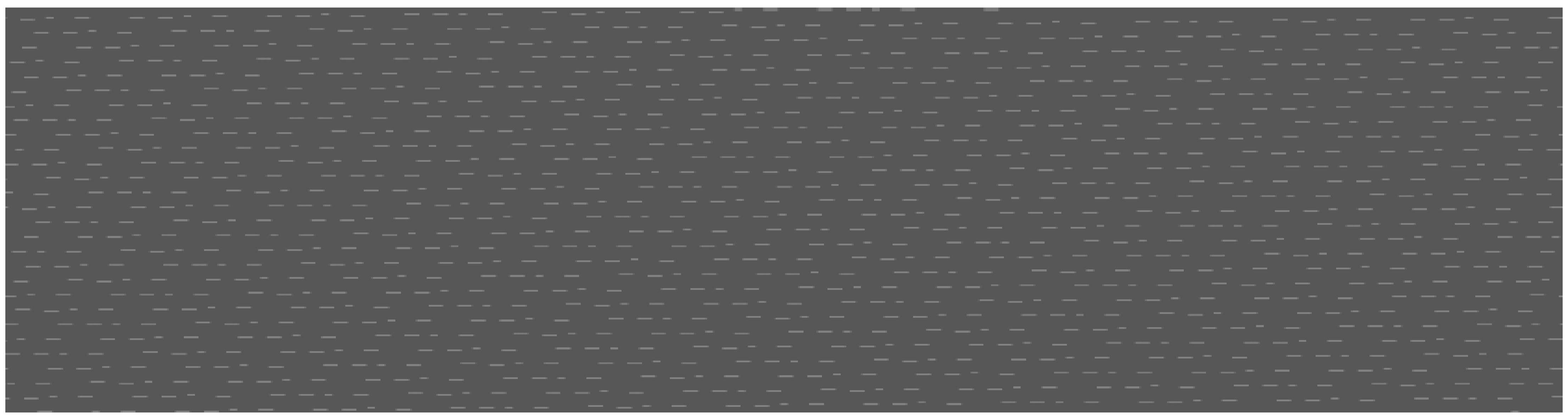


图 2-7　设置 JAVA_HOME

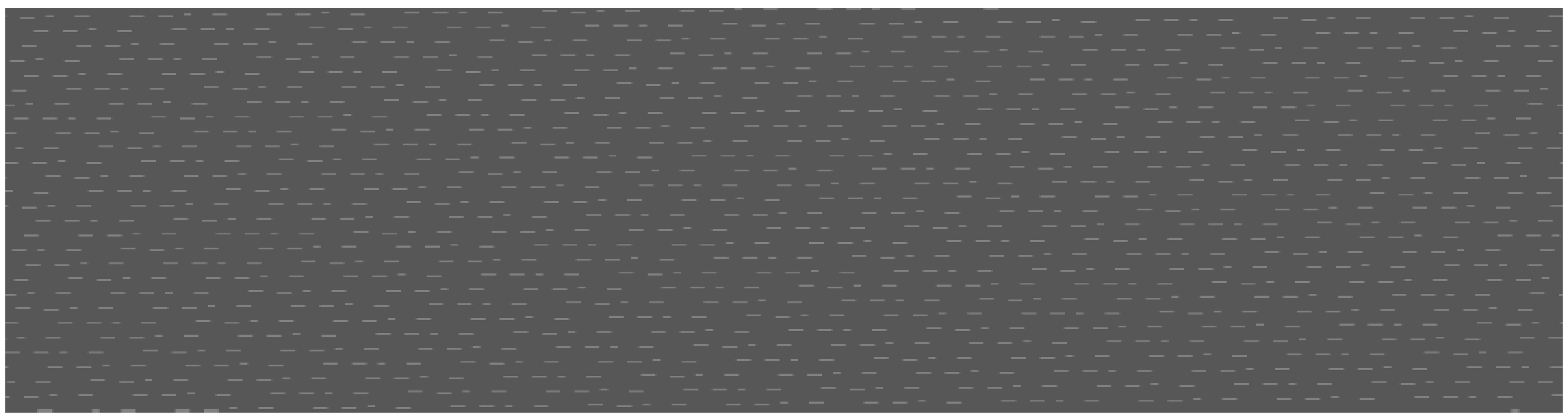


图 2-8　添加 Path 变量对话框

下面测试一下环境设置是否成功，可以通过在命令提示行中输入 javac 指令，看是否能够找到该指令，若弹出如图 2-9 所示的界面则说明环境设置成功。

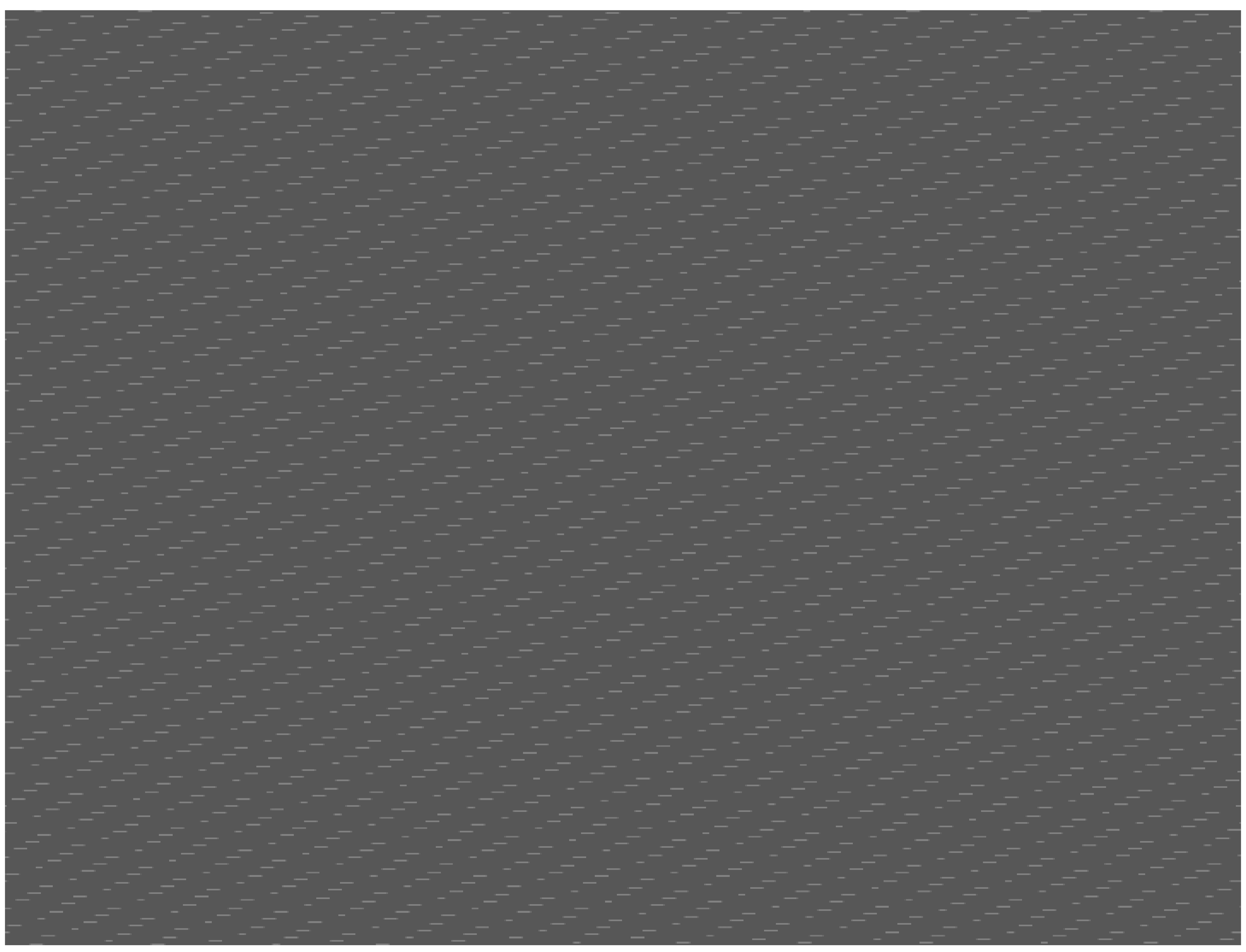


图 2-9　通过命令提示行测试环境变量

提示　打开命令行工具，也可以通过右击屏幕左下角的 Windows 图标，单击“命令提示符”菜单实现。

2.2　Eclipse 开发工具

Eclipse 是著名的跨平台 IDE 工具，最初 Eclipse 是 IBM 公司支持开发的免费 Java 开发工具，2001 年 11 月贡献给开源社区，现在它由非营利软件供应商联盟 Eclipse 基金会管理。Eclipse 本身也是一个框架平台，它有着丰富的插件，如 C++、Python、PHP 等开发其他语言的插件。另外，Eclipse 是绿色软件，不需要写注册表，卸载非常方便。

2.2.1　Eclipse 下载和安装

本书采用 Eclipse 4.6[①] 版本作为 IDE 工具，Eclipse 4.6 下载地址是 http://www.eclipse.org/downloads/。如图 2-10 所示是 Windows 系统下载 Eclipse 的页面，单击 DOWNLOAD 64 BIT 按钮，会跳转到如图 2-11 所示的选择下载镜像地址页面，单击 Select Another Mirror 链接可以改变下载镜像地址，然后单击 DOWNLOAD 按钮开始下载。

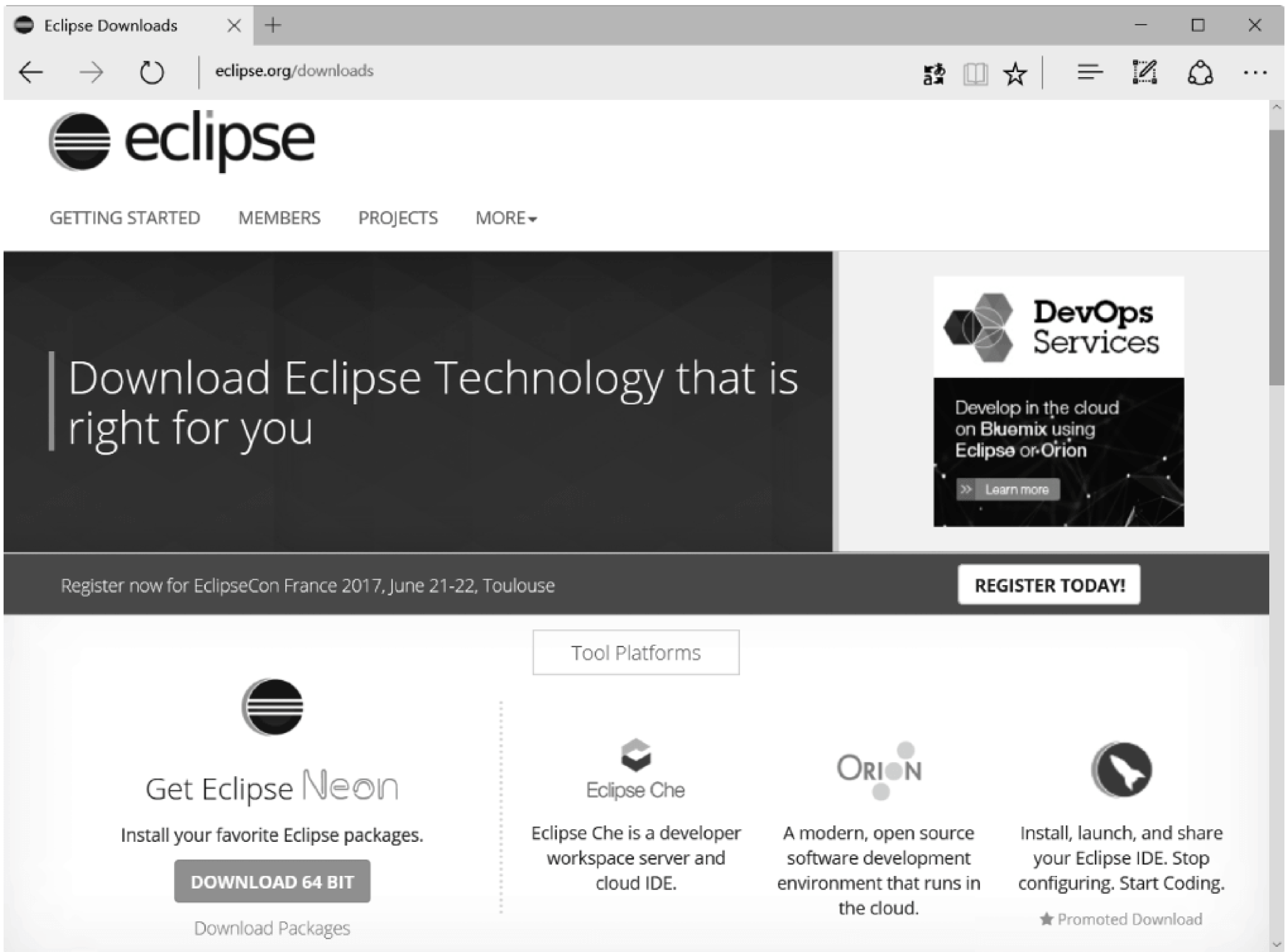


图 2-10　Eclipse 4.6 下载页面

下载完成后的文件是 eclipse-inst-win64.exe。事实上，eclipse-inst-win64.exe 是安装各种 Eclipse 版本客户端，双击 eclipse-inst-win64.exe 弹出如图 2-12 所示的界面，选择 Eclipse IDE for Java Developers 进入如图 2-13 所示的界面。在该界面中，通过 Installation

① Eclipse 4.6 开发代号是 Neon(氖气)，Eclipse 开发代号的首字母是按照字母顺序排列的。Eclipse 4.7 开发代号是 Oxygen(氧气)。

Folder 可以改变安装目录，选中 create start menu entry 单选按钮可以添加快捷方式到开始菜单，选中 create desktop shortcut 单选按钮可以在桌面创建快捷方式，设置完成后单击 INSTALL 按钮开始安装。安装完成后单击 LAUNCH 按钮可以启动 Eclipse，如图 2-14 所示。

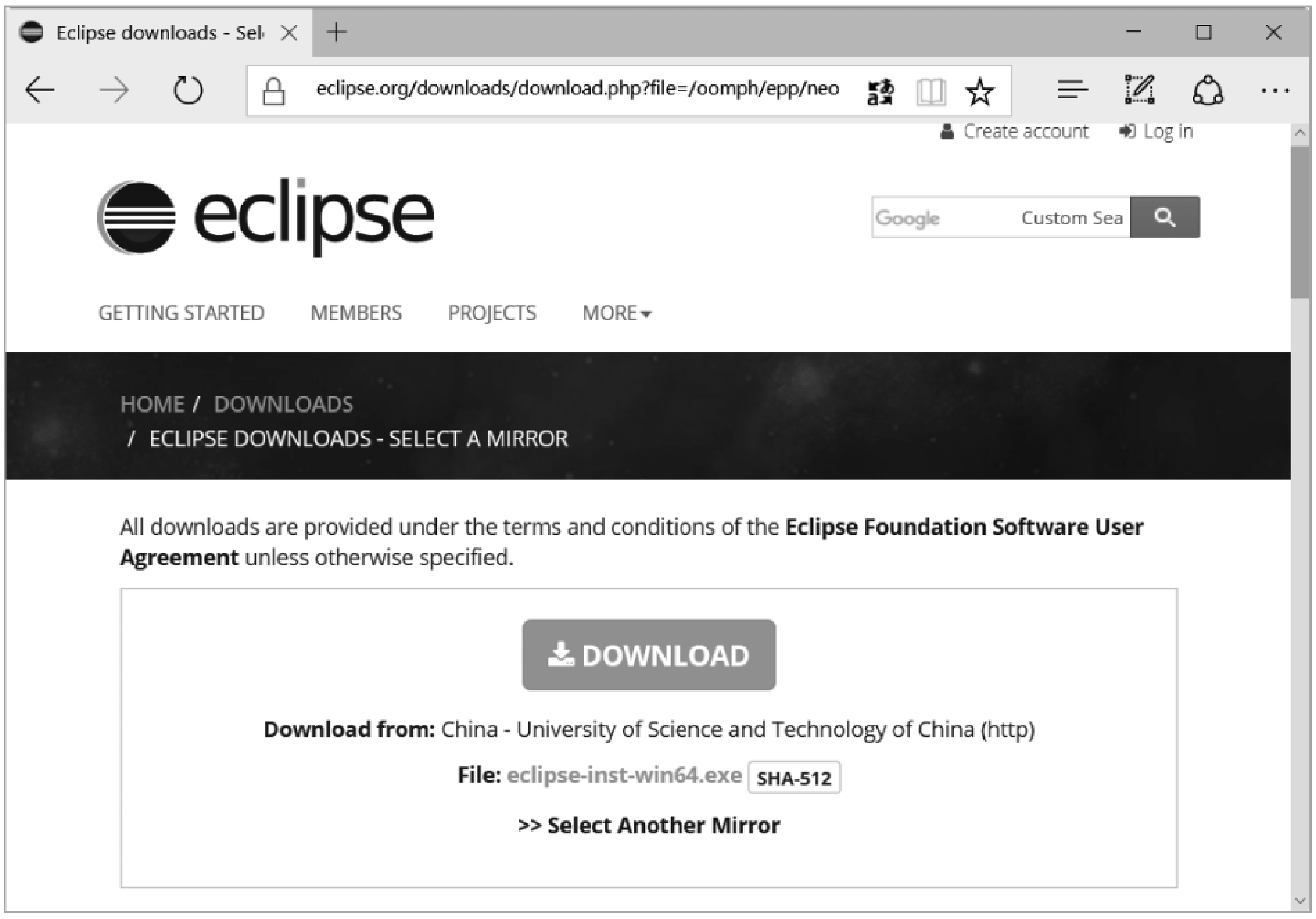


图 2-11 选择下载镜像地址



图 2-12 安装各种 Eclipse 版本客户端

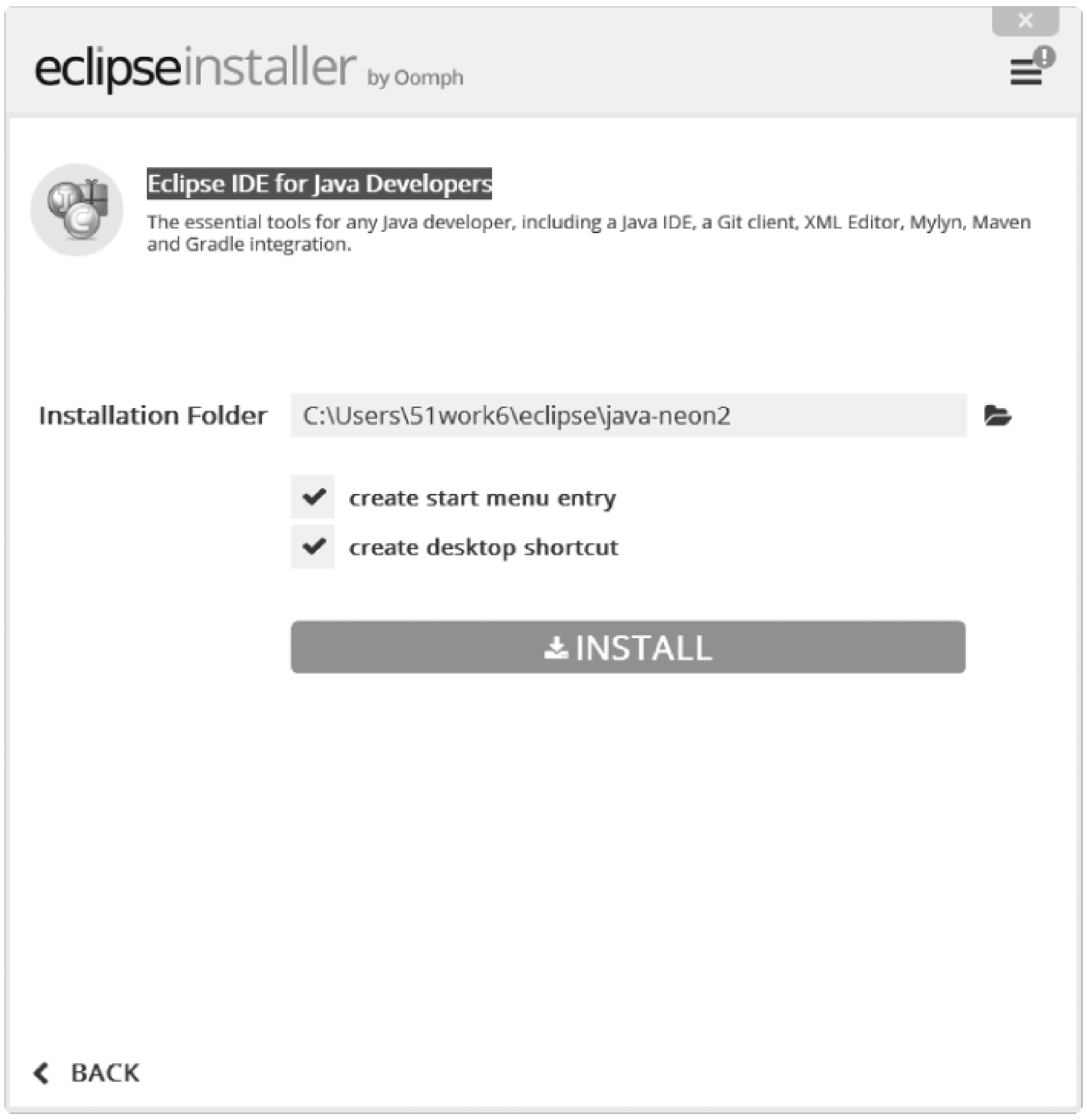


图 2-13　Eclipse 安装

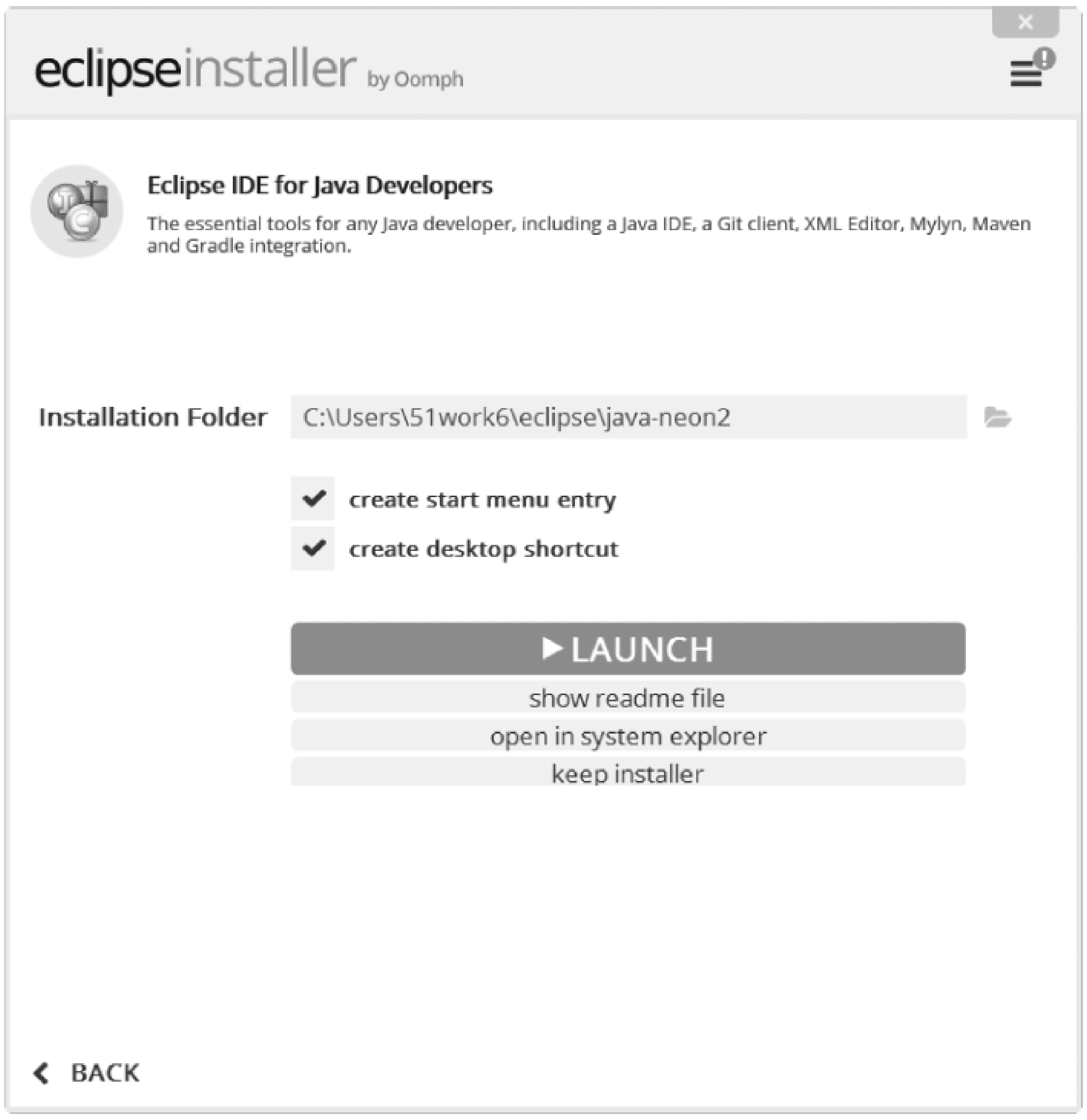


图 2-14　Eclipse 安装完成

在 Eclipse 启动过程中，会弹出如图 2-15 所示的选择工作空间(workspace)对话框，工作空间是用来保存工程的目录。默认情况下，每次 Eclipse 启动时都需要选择工作空间。如果觉得每次启动时都选择工作空间比较麻烦，则可以选中 Use this as the default and do not ask again 复选框，设置工作空间默认目录。初次启动 Eclipse 成功后，会进入如图 2-16 所示的欢迎界面。

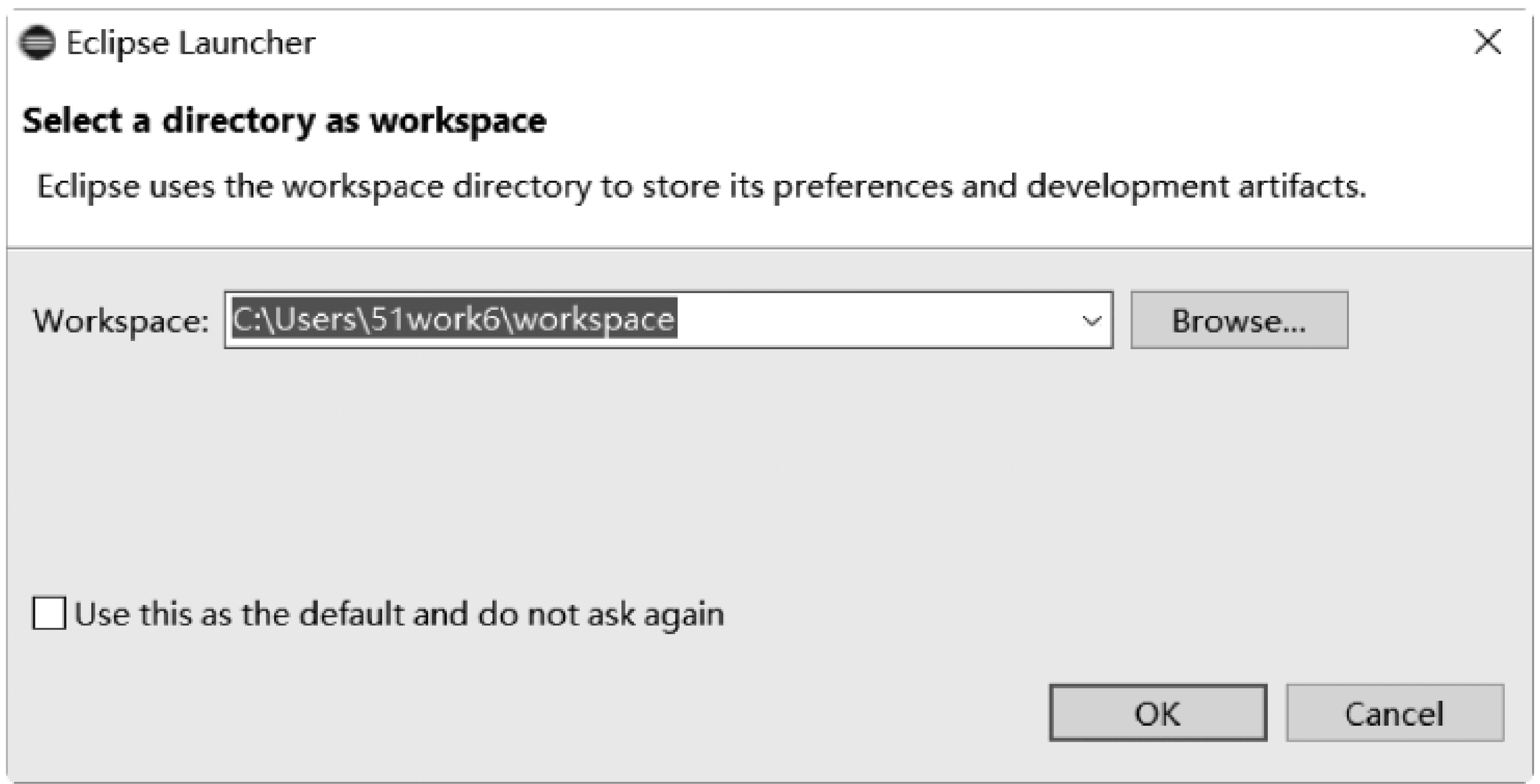


图 2-15　选择工作空间

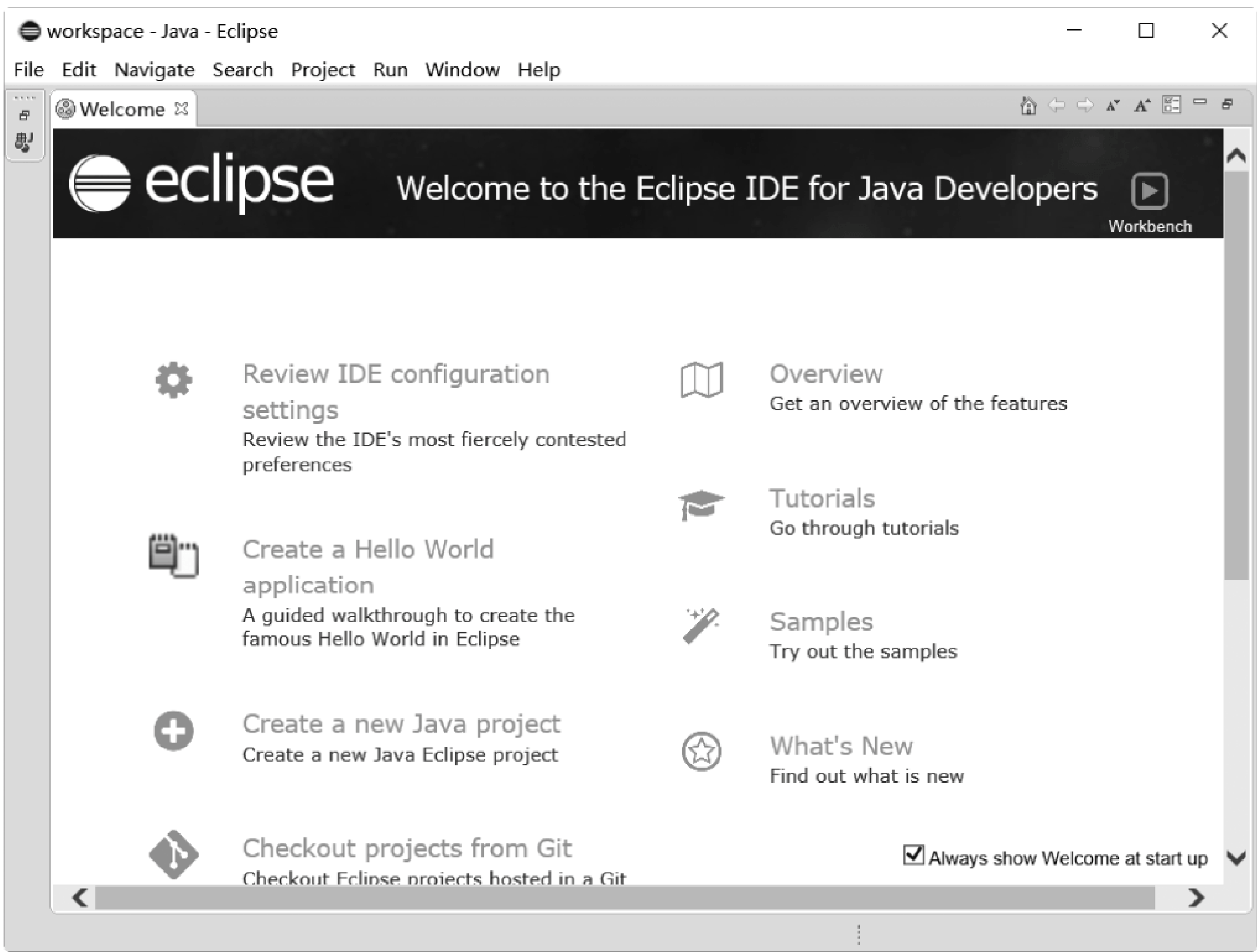


图 2-16　Eclipse 欢迎界面

2.2.2　安装中文语言包

Eclipse 界面默认是英文，对于一些初学者，英文界面使用起来还是有一定困难的。Eclipse 平台提供了一个语言包项目——Eclipse Babel Project(http://www.eclipse.org/babel/)，Babel 是一个插件，安装 Babel 插件可以通过离线或在线安装，Babel 插件下载地址是 http://www.eclipse.org/babel/downloads.php。如图 2-17 所示，单击 Zipped p2 repository for Neon 超链接下载离线包，注意离线包所支持的 Eclipse 版本。推荐在线安装，从图 2-17 所示页面中可见在线安装网址是 http://download.eclipse.org/technology/babel/update-site/R0.15.1/neon。

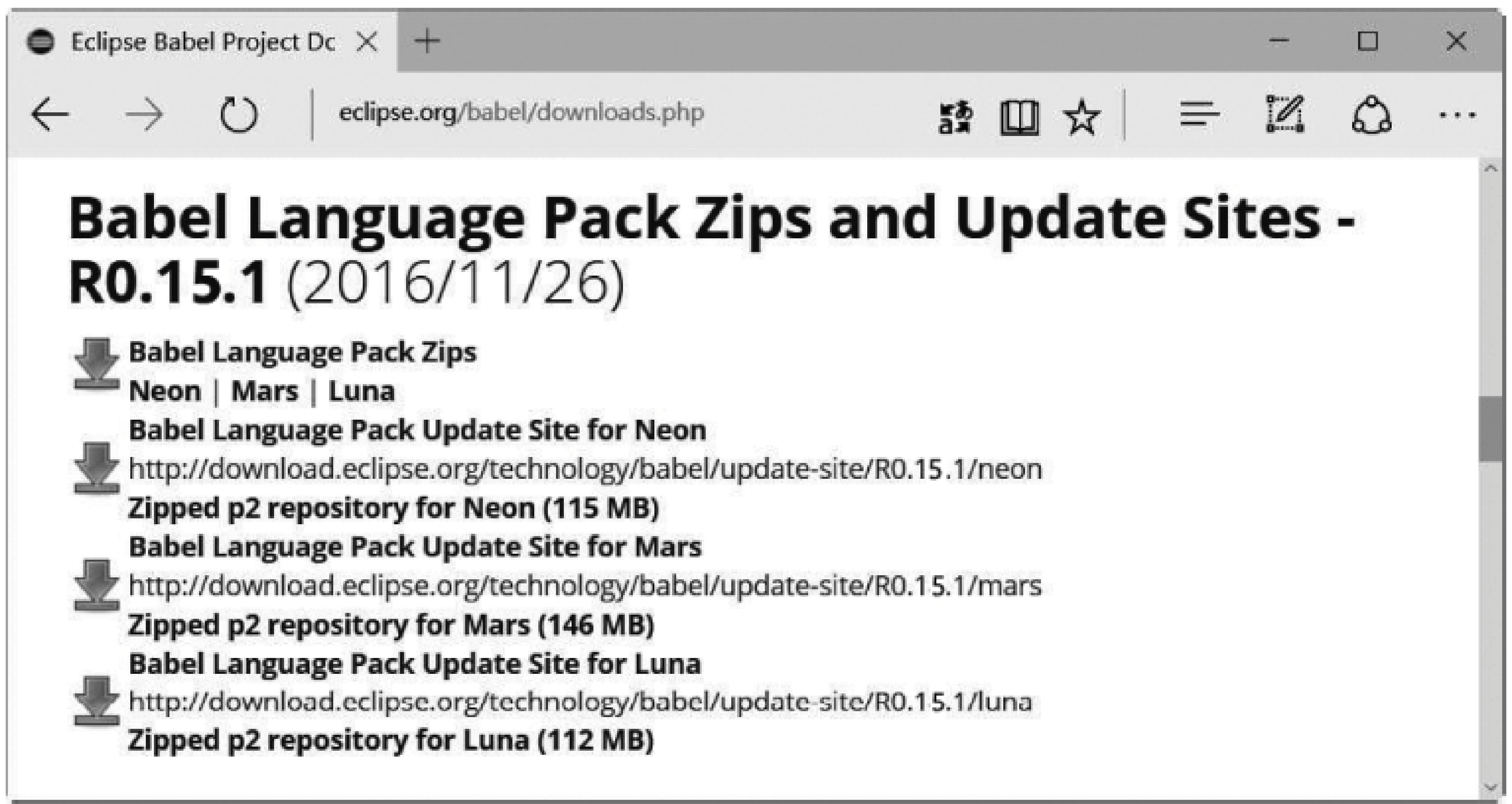


图 2-17　下载 Eclipse 语言包

安装插件过程为：首先启动 Eclipse，选择 Help→Install New Software 命令，系统弹出如图 2-18 所示的对话框。单击 Add 按钮弹出如图 2-19 所示对话框，在 Location 文本框中输入插件在线地址 http://download.eclipse.org/technology/babel/update-site/R0.15.1/neon，如图 2-20 所示。

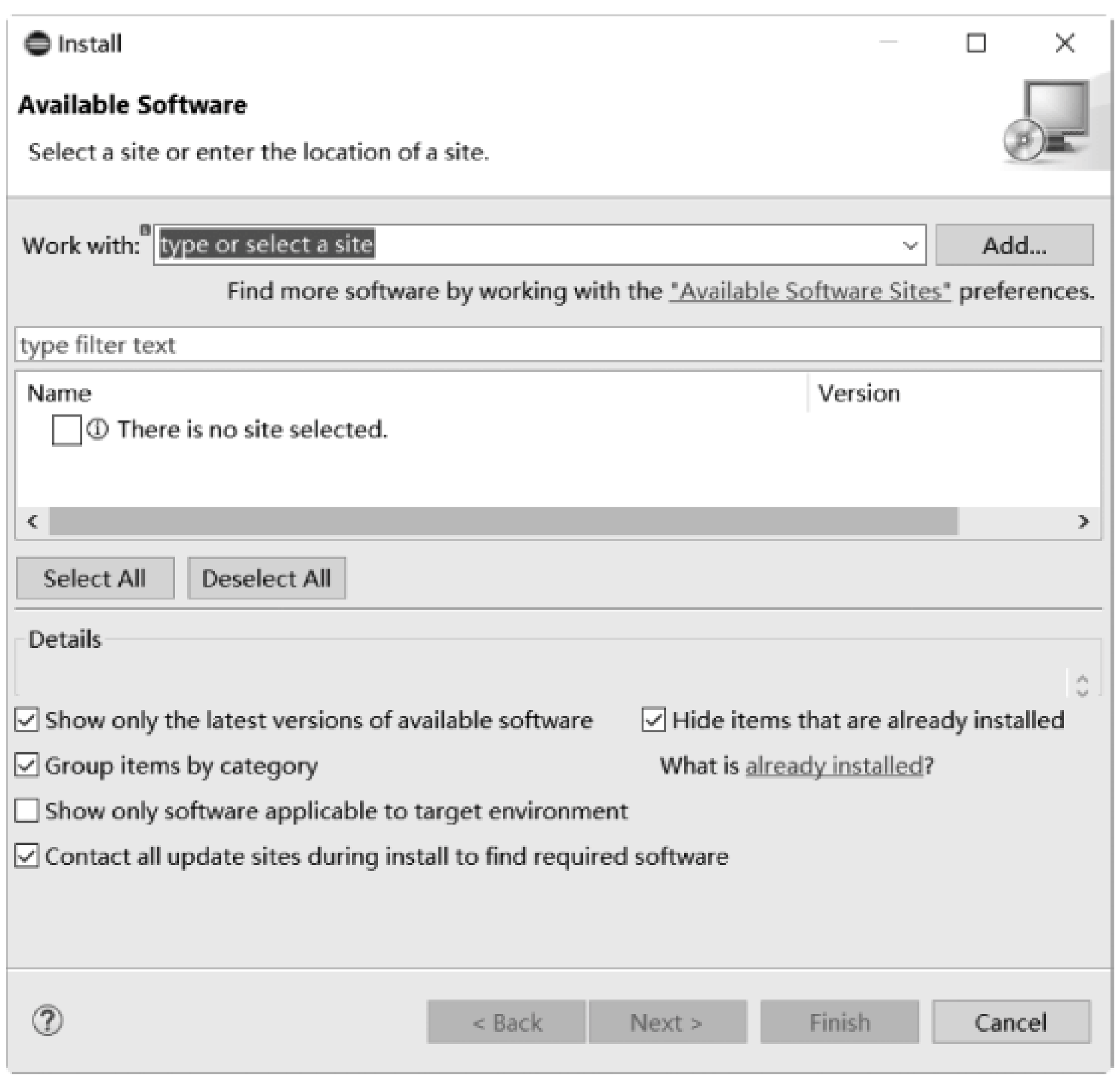


图 2-18　安装插件

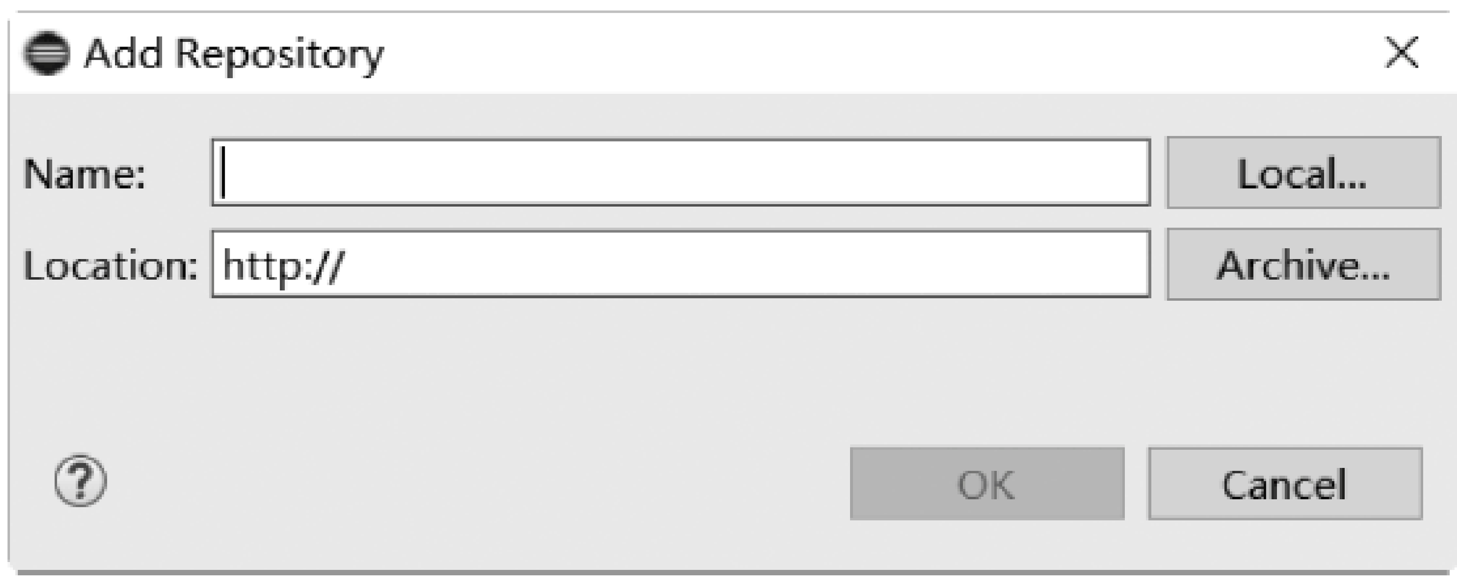


图 2-19　插件地址

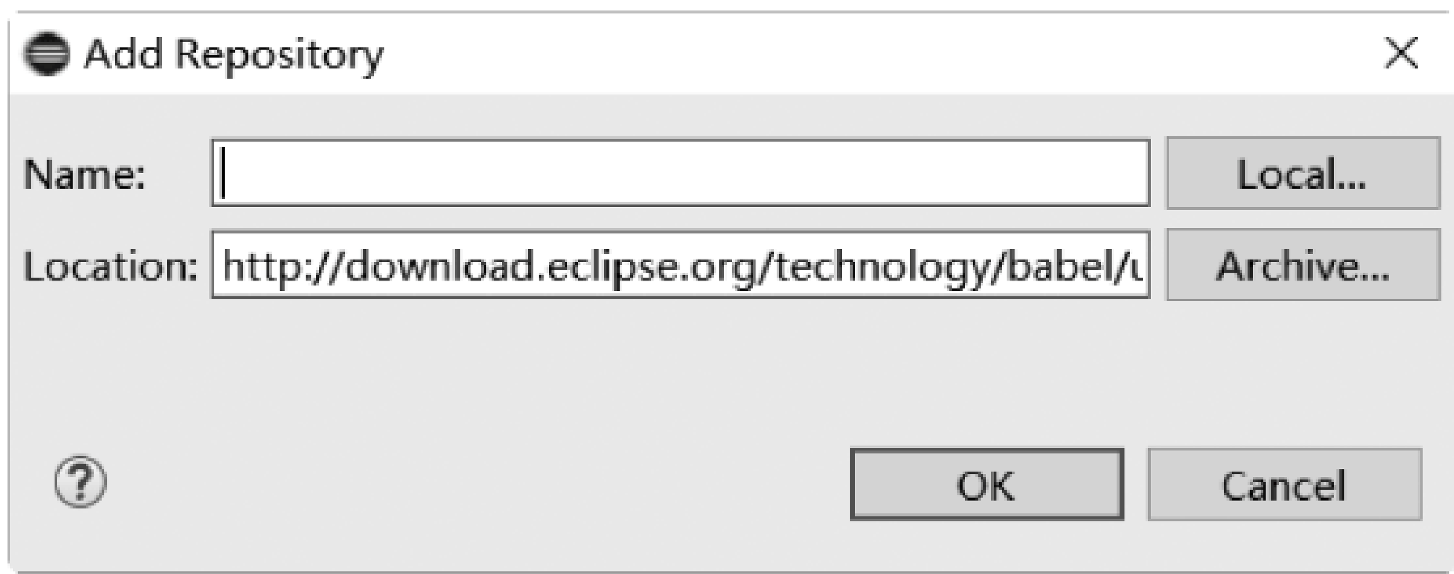


图 2-20　输入插件地址

确定输入内容后单击 OK 按钮关闭对话框，Eclipse 通过刚刚输入的网址查找插件，如果能够找到插件，则弹出如图 2-21 所示的对话框，从中选择简体中文语言包。选择完成后单击 Next 按钮进行安装，安装过程中需要从网上下载插件，这个过程需要等一段时间。

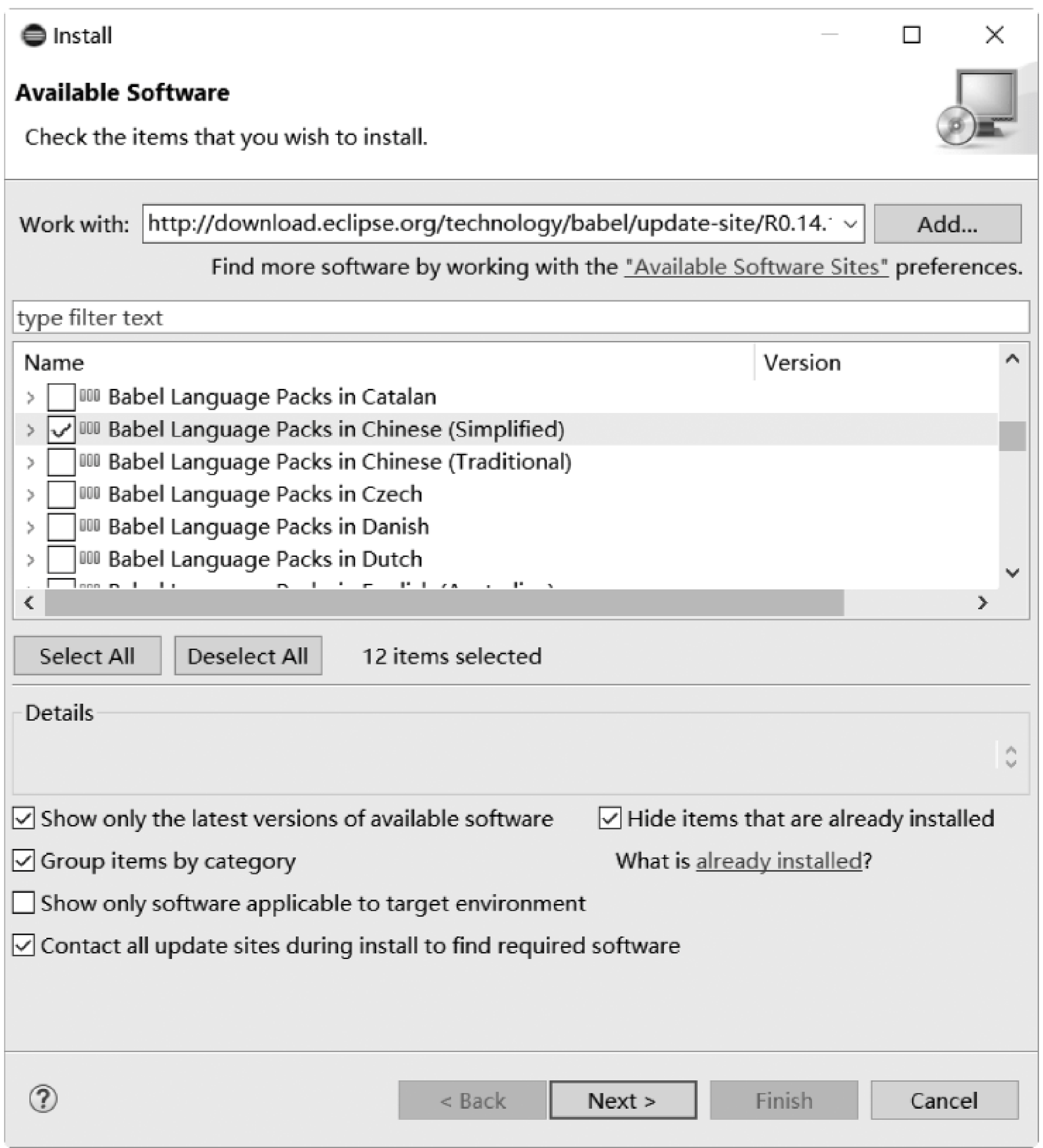


图 2-21　选择简体中文语言包

安装简体中文语言包插件后重新启动 Eclipse，界面如图 2-22 所示。

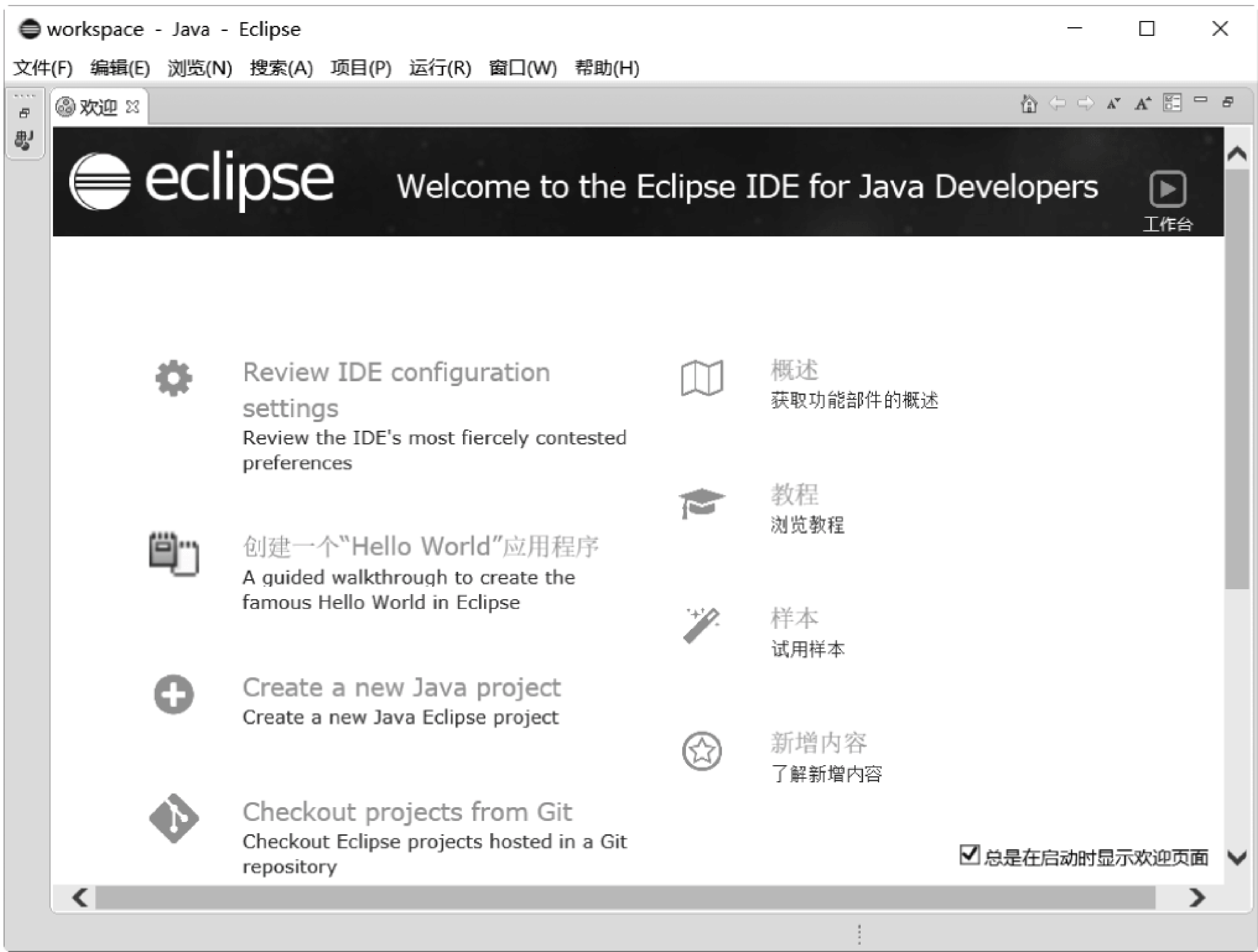


图 2-22　安装简体中文语言包后的 Eclipse 界面

2.2.3　Eclipse 界面

关闭 Eclipse 的"欢迎"界面，并创建一个 Java 工程后（如何创建 Java 工程将在第 3 章介绍），可以看到如图 2-23 所示的主界面。该界面主要分成 4 个区域。

①号区域是包资源管理器视图，以包形式管理 Java 源文件。包是一种命名空间，将在后面详细介绍。

②号区域是代码编辑视图，编码工作就是在这里完成的。

③号区域是显示大纲等辅助视图，大纲视图中列出了当前 Java 类中方法和成员变量，并且单击可以快速导航到指定代码。

④号区域是显示问题、控制台等辅助视图，可以列出当前工程的编译错误和警告等信息。

事实上，这 4 个区域视图都可以互换，只要拖曳视图标题到相应的区域即可完成互换。Eclipse 视图标题如图 2-24 所示，标题的右端有两个按钮：最小化按钮和最大化按钮，单击可以实现视图的最小化和最大化显示。

此外，Eclipse 提供了丰富的菜单和工具栏，随着学习的深入，本书会有重点地加以介绍，这里不再赘述。

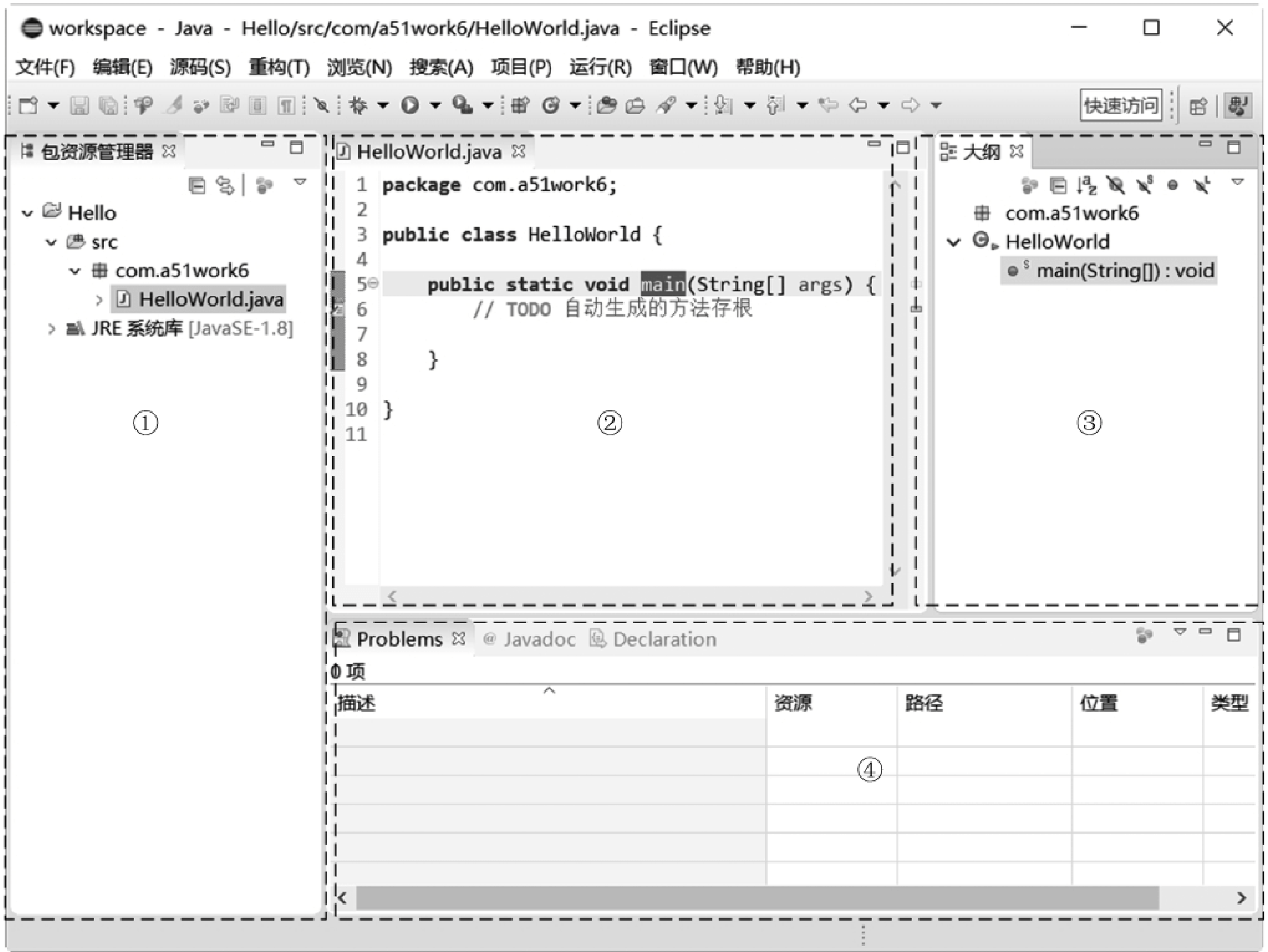


图 2-23　Eclipse 主界面

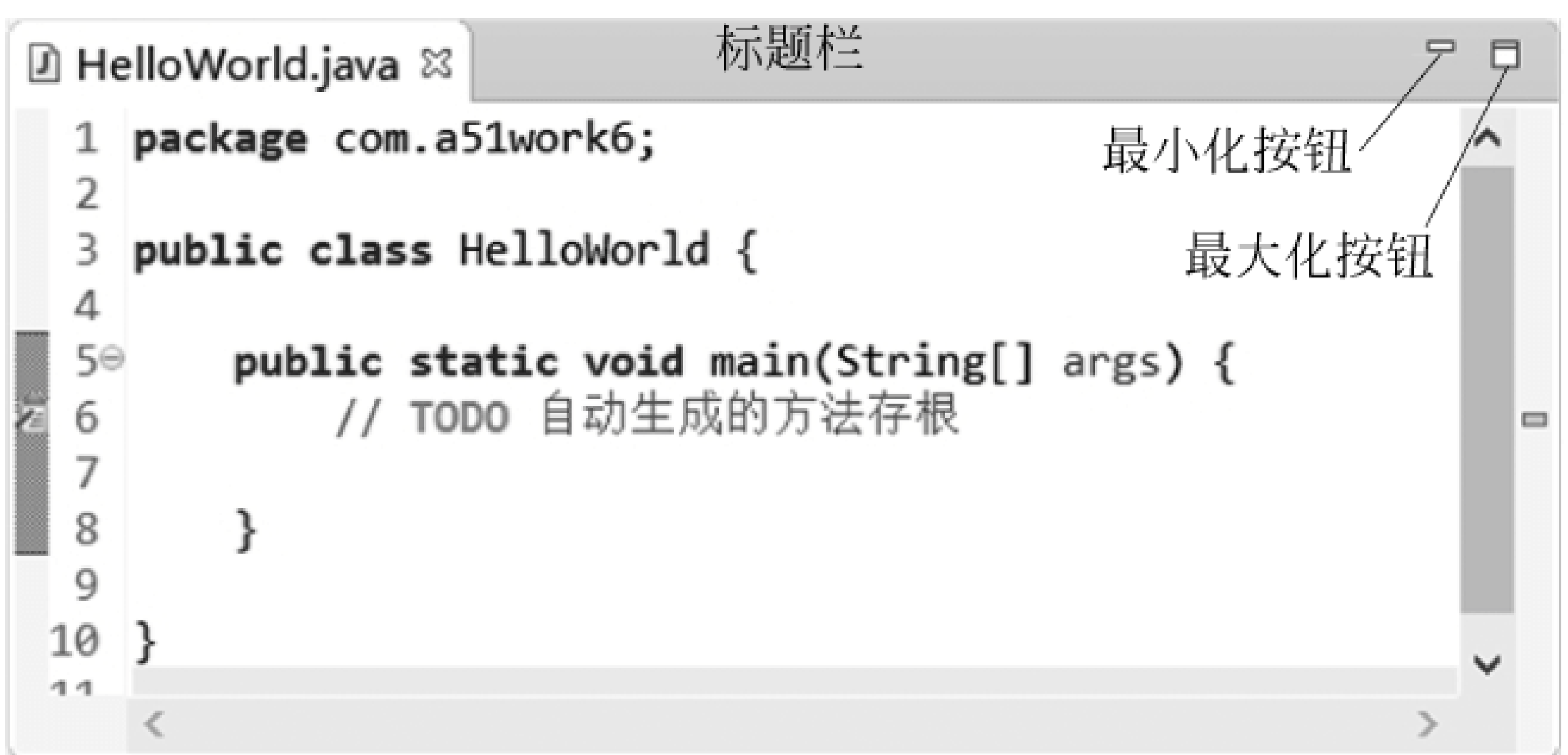


图 2-24　Eclipse 视图

2.2.4　Windows 系统中常用快捷键

一个优秀的 IDE 开发工具应该提供丰富的快捷键，快捷键虽然不能完全替代鼠标操作，但却可以锦上添花。由于 Eclipse 工具提供很多快捷键，本书不打算介绍全部的快捷键，笔者总结了一些 Eclipse 工具在 Windows 系统常用的快捷键，如表 2-1 所示。

表 2-1　Eclipse 在 Windows 系统常用快捷键

作　用　域	快　捷　键	功　　能
全局	Ctrl+M	最大化/最小化当前视图
全局	Ctrl+=	放大视图
全局	Ctrl+-	缩小视图
文本编辑器	Ctrl+F	查找并替换
文本编辑器	Ctrl+L	转至某行
Java 编辑器	Ctrl+Shift+F	代码格式化
Java 编辑器	Ctrl+/	注释/取消注释当前行
Java 编辑器	Ctrl+Shift+M	添加导入包
Java 编辑器	Ctrl+Shift+O	组织导入包
Java 编辑器	Ctrl+Shift+↑	转至上一个成员
Java 编辑器	Ctrl+Shift+↓	转至下一个成员
Java 编辑器	Ctrl+B	重新编译 Java 程序代码
Java 编辑器	Ctrl+F11	运行上次程序

这些快捷键只是冰山一角，想了解更多 Eclipse 在 Windows 系统常用快捷键，可以参考 http://baike.baidu.com/item/Eclipse 快捷键指南。

2.3　其他开发工具

Java IDE 开发工具除了 Eclipse 当然还有很多，其中被广泛认可的还有 IntelliJ IDEA 和 NetBeans，令人惊奇的是它们都源自捷克人之手。

2.3.1　IntelliJ IDEA

虽然 IntelliJ IDEA 市场份额不如 Eclipse，但是被很多 Java 专家认为是最优秀的 Java IDE 开发工具。IntelliJ IDEA 是 Jetbrains 公司（www.jetbrains.com）研发的一款 Java IDE 开发工具。Jetbrains 是一家捷克公司，该公司开发的很多工具都好评如潮。如图 2-25 所示为 Jetbrains 开发的工具，这些工具可以编写 C/C++、C#、DSL、Go、Groovy、Java、JavaScript、Kotlin、Objective-C、PHP、Python、Ruby、Scala、SQL 和 Swift 语言。

IntelliJ IDEA 下载地址是 https://www.jetbrains.com/idea/download/，从图 2-26 所示页面可见，IntelliJ IDEA 有两个版本：Ultimate（旗舰版）和 Community（社区版）。旗舰版是收费的，可以免费试用 30 天，如果超过 30 天，则需要购买软件许可（License key）。社区版是完全免费的，对于学习 Java 语言，社区版已经足够了。在图 2-26 所示页面下载完 IntelliJ IDEA 工具，即可安装。

IntelliJ IDEA 工具使用起来比较复杂，而且用户群少，因此 IntelliJ IDEA 具体使用细节本书不再介绍。

2.3.2　NetBeans IDE

NetBeans 是一个始于捷克布拉格查理大学的学生项目（Xelfi 计划），Xelfi 计划延伸发展成为 NetBeans IDE 工具，1999 年被 Sun 公司收购，后来随着 Oracle 公司收购 Sun 公司，

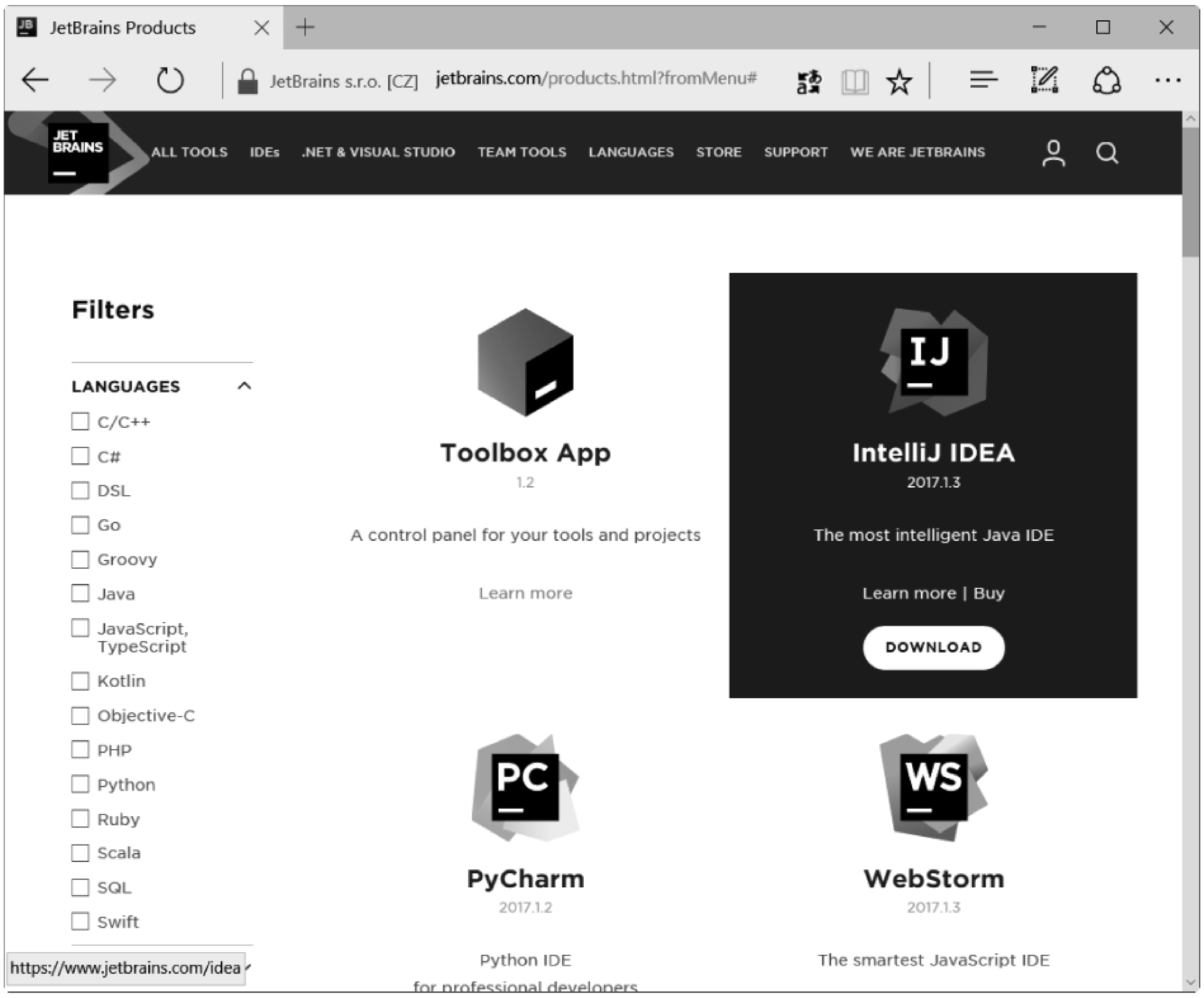


图 2-25　Jetbrains 公司开发的工具

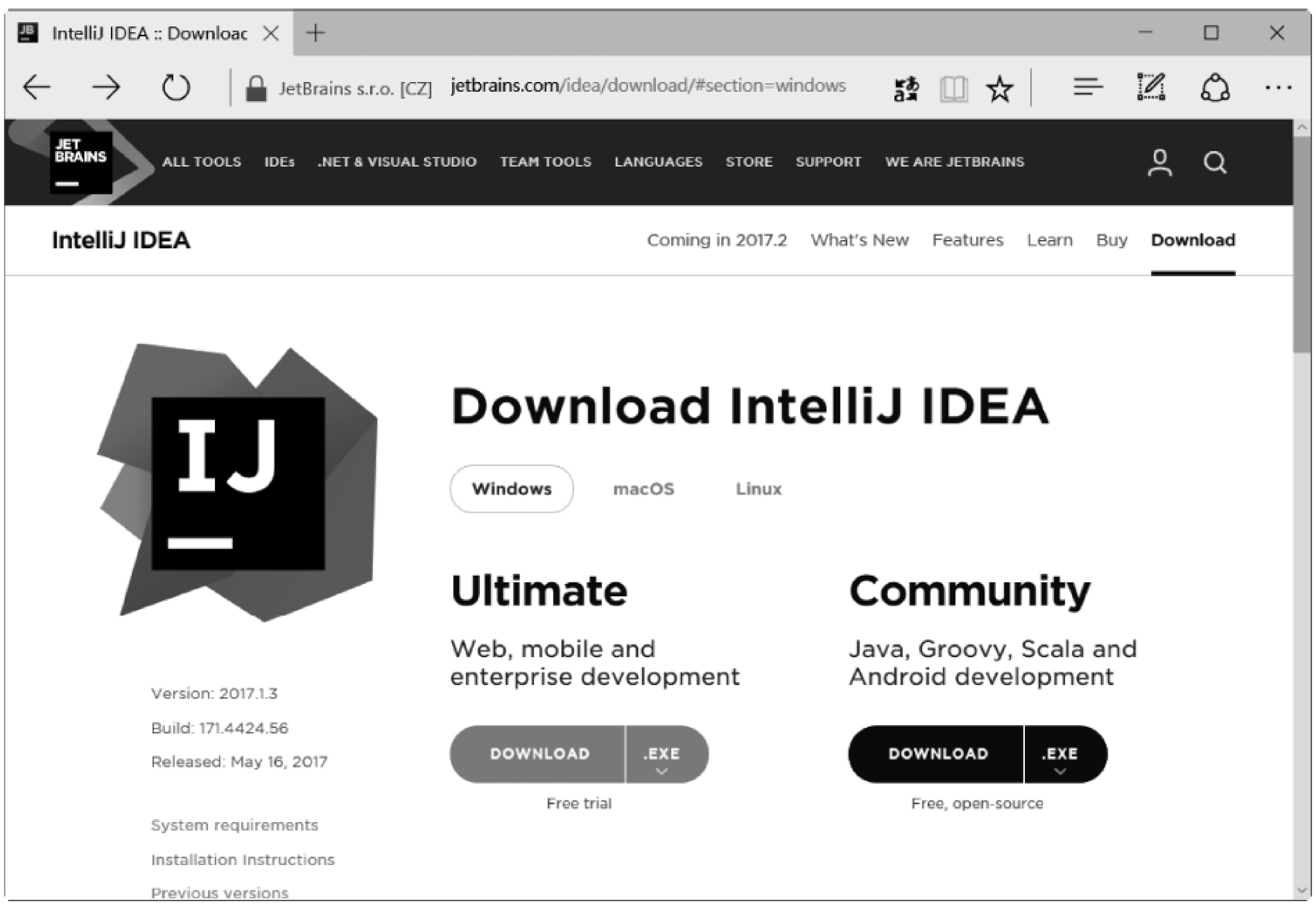


图 2-26　下载 IntelliJ IDEA

NetBeans IDE 成为 Oracle 工具产品。

被 Oracle 收购后 NetBeans IDE 仍然是免费工具，下载网址为 https://netbeans.org/downloads/，打开页面如图 2-27 所示。NetBeans IDE 支持的平台有 Windows、Mac OS X 和 Linux 等，除完全支持所有 Java 平台（Java SE、Java EE、Java ME 和 Java FX）之外，还支持 PHP、HTML5、JavaScript、Groovy 和 C/C++等语言。在图 2-27 所示页面选择适合自己的版本下载 NetBeans IDE，完成之后即可安装。

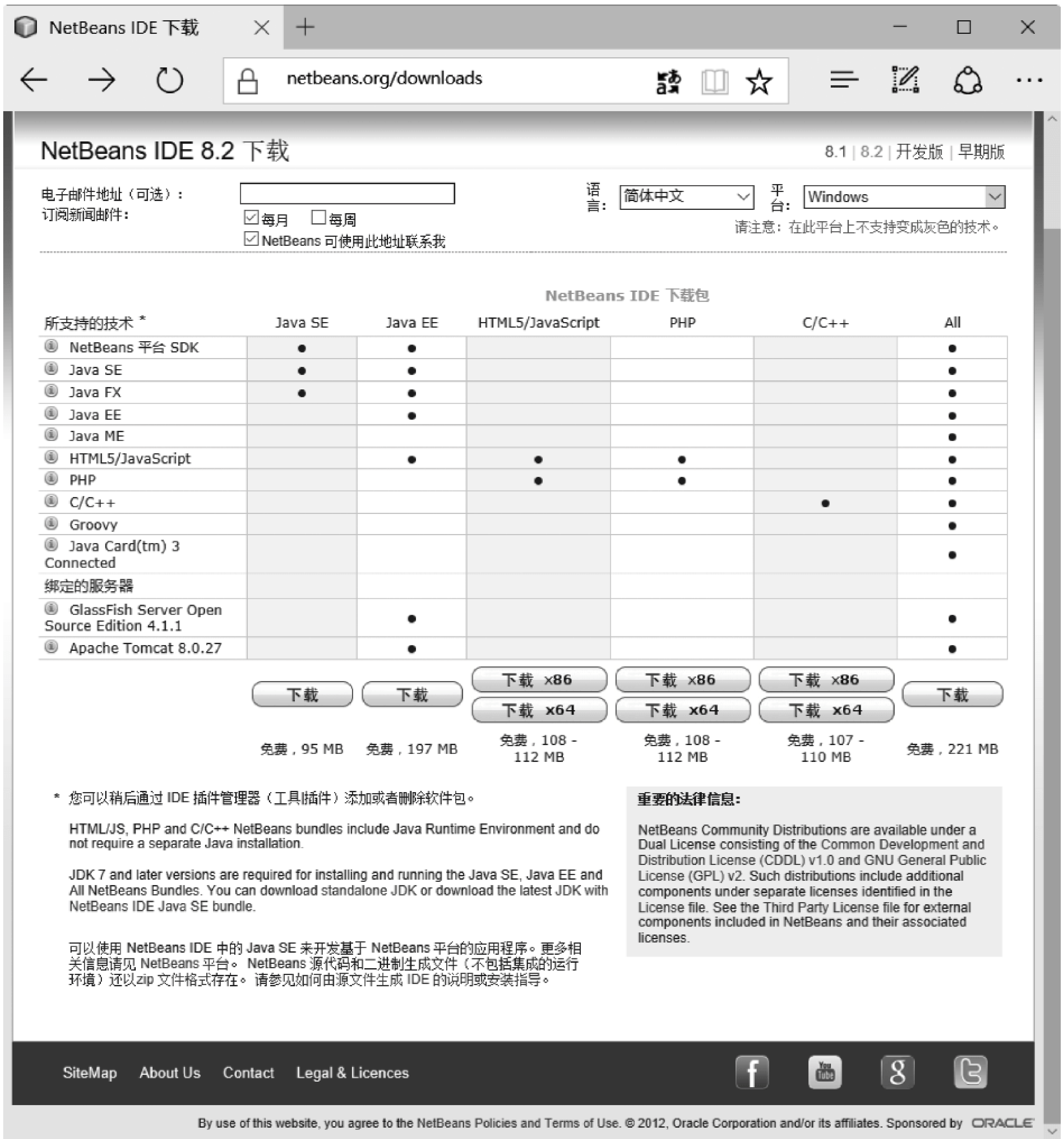


图 2-27　NetBeans IDE 下载页面

NetBeans IDE 用户群比较少，因此 NetBeans IDE 具体使用细节本书不再介绍。

2.3.3　文本编辑工具

IDE 开发工具提供了强大的开发能力与语法提示功能，但对于学习 Java 的学员而言，语法提示并不是件好事，建议初学者采用文本编辑工具＋JDK 学习。开发过程就使用文本编辑工具编写 Java 源程序，然后使用 JDK 提供的 javac 指令编译 Java 源程序，再使用 JDK

和 JRE 提供的 java 指令运行。

提示 javac 和 java 等指令需要在命令提示行中执行，打开命令行参考 2.1.2 节。

Windows 平台下的文本编辑工具有很多，常用如下。

- 记事本：Windows 平台自带的文本编辑工具，关键字不能高亮显示。
- UltraEdit：历史悠久、强大的文本编辑工具，可支持文本列模式等很多有用的功能，官网 www.ultraedit.com。
- EditPlus：历史悠久、强大的文本编辑工具，小巧、轻便、灵活，官网 www.editplus.com。
- Sublime Text：近年来发展和壮大的文本编辑工具，所有的设置没有图形界面，在 JSON 格式[①]的文件中进行，初学者入门比较难，官网 www.sublimetext.com。

除了记事本工具外，其他的 UltraEdit、EditPlus 和 Sublime Text 等工具都可以与 JDK 集成起来，能够在这些工具中直接执行 JDK 指令。

下面重点介绍一下 EditPlus 与 JDK 集成过程。首先，打开 EditPlus，选择“工具”→“首选项”命令，弹出首选项对话框，如图 2-28 所示，选择“工具”→“自定义工具”，在“自定义工具组及项目”列表框中选择 Group 1。然后通过下面的步骤添加编译和执行菜单。



图 2-28 EditPlus 设置参数

1. 添加编译菜单

在图 2-28 所示界面单击“添加工具”→“程序”按钮，添加一个命令菜单。如图 2-29 所示，输入并选择相关项目，其中“菜单文本”文本框中输入的是出现在“工具”菜单中的菜单名，这里可以根据自己喜好取名字；“命令”是菜单要执行的 JDK 指令，这里指定 JDK 中 javac.exe 文件路径；“参数”是指命令后面的参数，这里需要指定要编译的文件名，

① JSON(JavaScript Object Notation，JS 对象标记)是一种轻量级的数据交换格式，采用键值对形式，如{"firstName"："John"}。

$(FileName)是 EditPlus 获得文件名的系统变量，$(FileName)是带有扩展名的文件名；“起始目录”是命令执行的目录，$(FileDir) 是 EditPlus 获得文件当前文件目录的系统变量；最后还需要在“动作”中选择“捕获控制台输出”，以便将命令执行结果输出到 EditPlus 控制台。



图 2-29　添加编译菜单

2. 添加执行菜单

参考“添加编译菜单”的添加过程，添加一个命令菜单。如图 2-30 所示，在“命令”文本框中指定 JDK 中 java.exe 文件路径；“参数”是 $(FileNameNoExt)，表示不带扩展名的文件名。

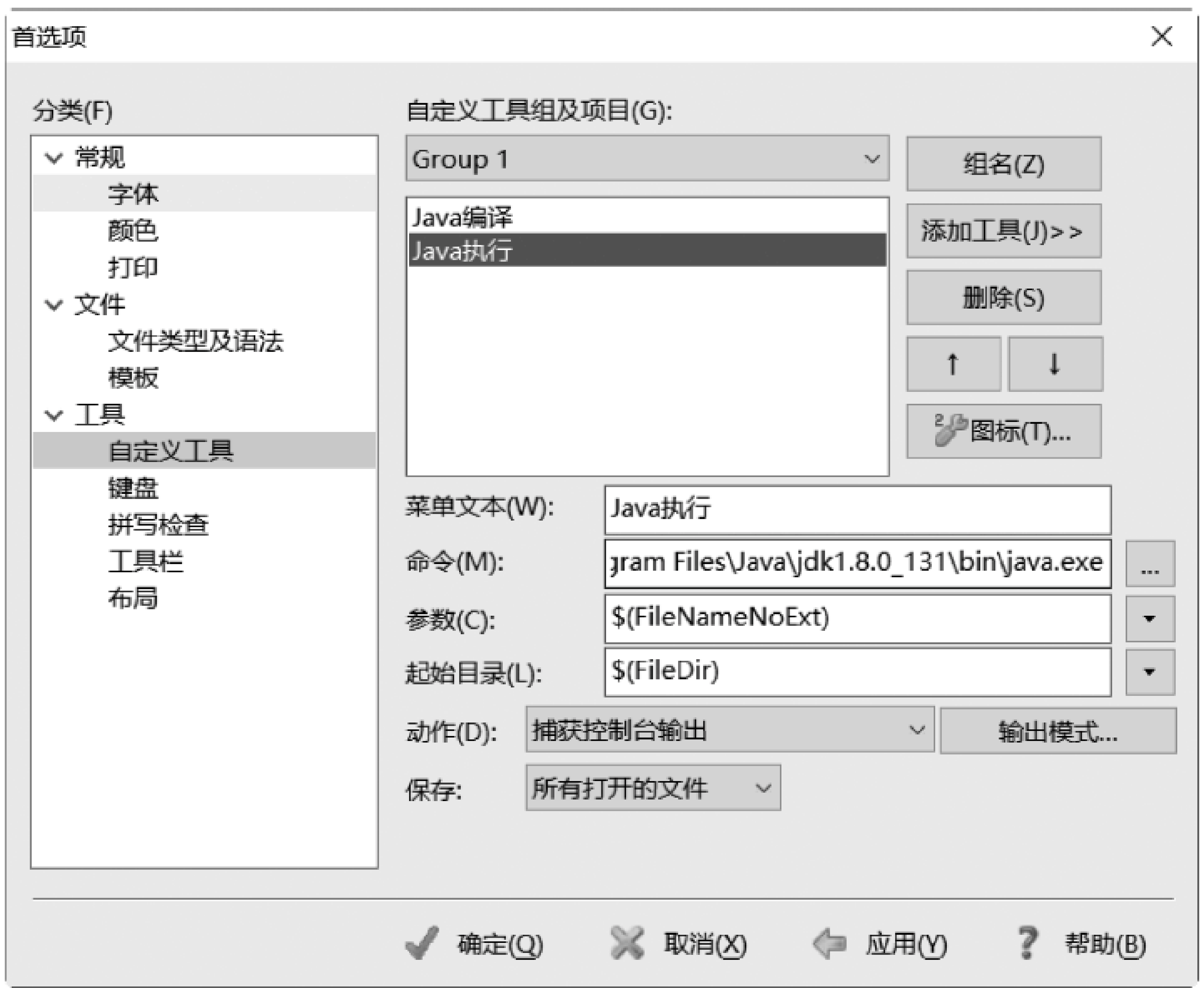


图 2-30　添加执行菜单

注意 编译时指定的 Java 源代码文件要带有扩展名，指令类似于 javac HelloWorld .Java。而运行时不需要指定字节码文件的扩展名，指令类似于 Java HelloWorld。

添加成功后会发现 EditPlus 的工具菜单中多出了两个子菜单，即 Java 编译和 Java 执行，如图 2-31 所示。当打开一个源程序 HelloWorld. java 时，可通过单击 Java 编译菜单(或按 Ctrl+1 快捷键)编写 HelloWorld. java，如图 2-32 所示，编译结果输出到 EditPlus 控制台；然后通过单击 Java 执行菜单(或按 Ctrl+2 快捷键)执行编译完成的字节码文件 HelloWorld. class，如图 2-33 所示，运行结果输出到 EditPlus 控制台。



图 2-31 添加后的工具菜单

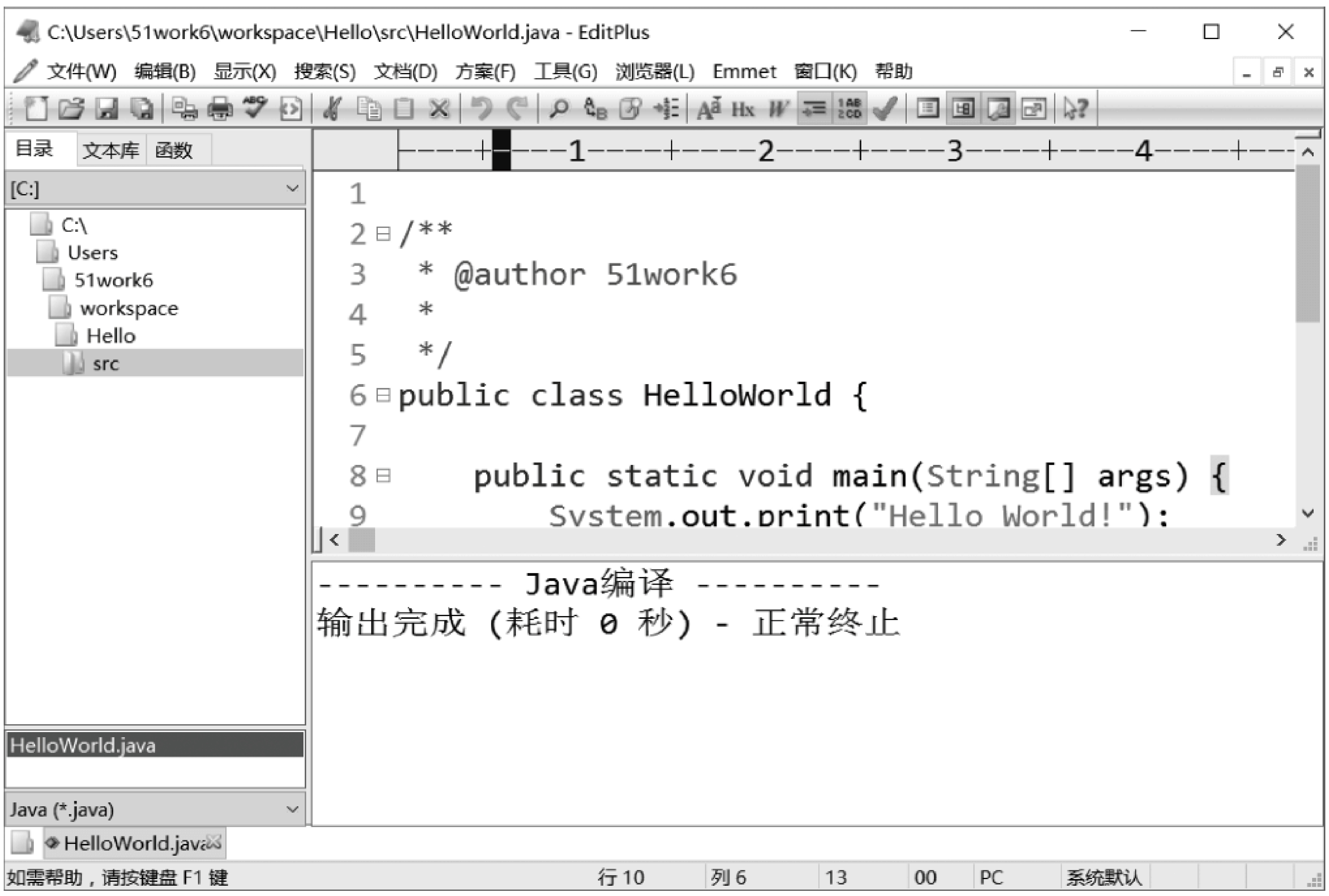


图 2-32 执行 Java 编译菜单

每一种文本编辑工具的配置方式都有很大差别，这里笔者不能一一穷尽，其他工具的配置过程读者可以参考工具的官方资料。

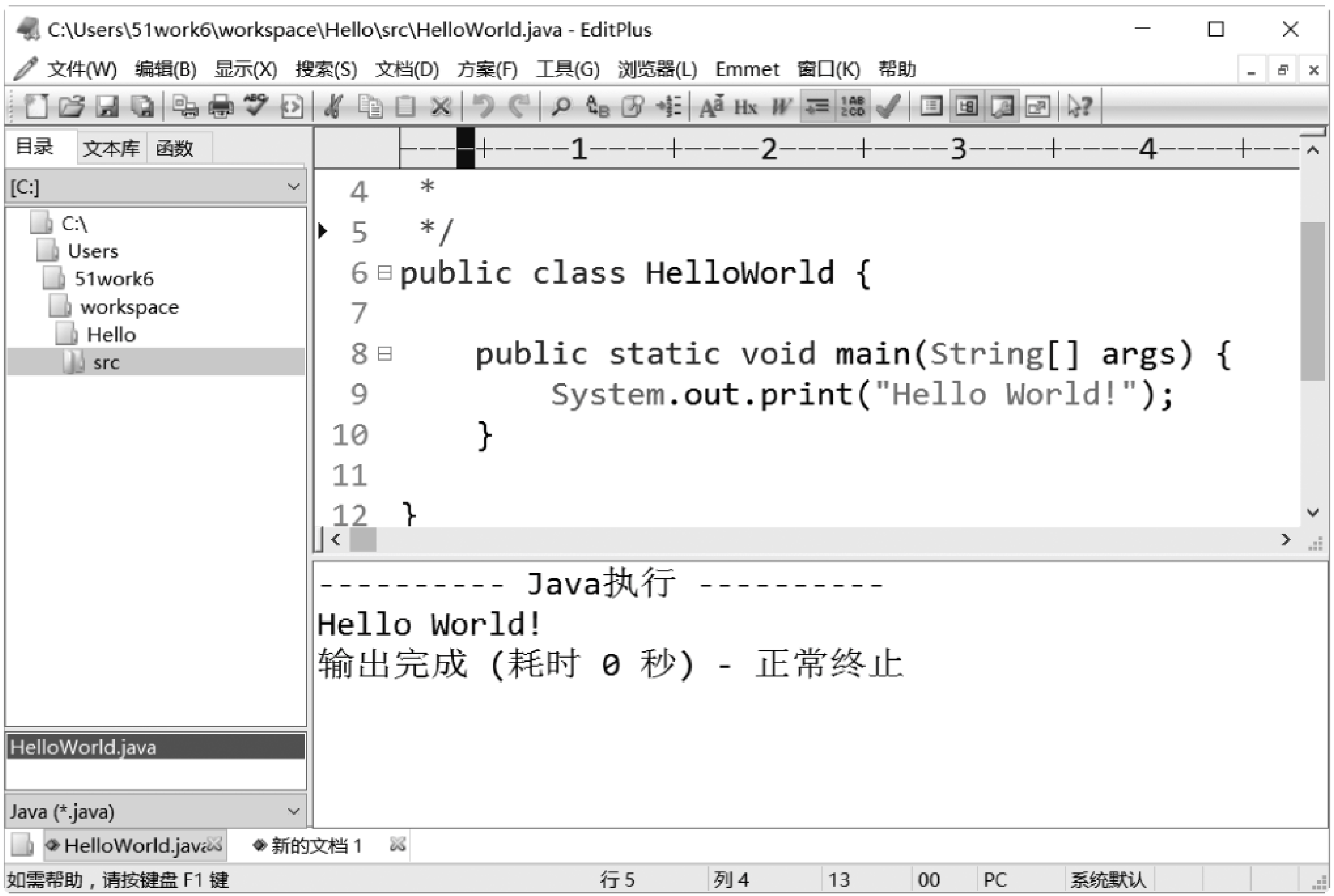


图 2-33　执行 Java 执行菜单

本章小结

通过对本章的学习，读者可以了解 Java 开发工具，其中重点是 Eclipse 工具的下载、安装和使用。此外，还介绍了其他的一些工具：IntelliJ IDEA 和 NetBeans，以及文本编辑工具 EditPlus+JDK 的配置过程。

2.4　同步练习

1. 在 Windows 平台安装和配置 JDK。
2. 在 Windows 平台安装和配置 Eclipse 开发工具。

第 3 章 第一个 Java 程序

CHAPTER 3

本书第一个 Java 程序是通过控制台输出 HelloWorld,以这个示例为切入点,系统介绍 Java 程序的编写、Java 源代码结构以及一些基础知识。

在 Java 中,程序都是以类的方式组织的,Java 源文件都保存在.java 文件中。每个可运行的程序都是一个类文件,或者称为字节码文件,保存为.class 文件。要实现在控制台中输出 HelloWorld 示例,则需要编写一个 Java 类。

3.1 使用 Eclipse 实现

HelloWorld 示例可通过多种工具实现,本节首先介绍如何通过 Eclipse 实现。

3.1.1 创建项目

在 Eclipse 中通过项目(Project)管理 Java 类,因此需要先创建一个 Java 项目,然后在项目中创建一个 Java 类。

Eclipse 创建项目步骤:打开 Eclipse,选择"文件"→"新建"→"Java 项目"命令,打开"新建 Java 项目"对话框,如图 3-1 所示。

下面简要说明图 3-1 中的各个选项。

- 项目名:指要创建的项目名称。
- 使用缺省位置:选中该复选框,创建的项目会保存到工作空间中。
- JRE:开发人员可以在这里指定项目运行所需要的 JRE,默认是使用系统 Path 环境变量所指定的 JRE。
- 项目布局:是设置项目中源文件和类文件的存放目录,默认情况下选中"为源文件和类文件创建单独的文件夹"单选按钮,这个选项被选中后,源文件和类文件会在两个不同的文件夹下,即源文件被放置在当前项目的文件夹中,类文件被放置在当前项目的 bin 文件夹中;如果选中"使用项目文件夹作为源文件和类文件的根目录"单选按钮,则源文件和类文件都被放置在当前项目根目录下,而且混合在一起。
- 工作集:可以将多个相关的项目集中在一个工作集中管理。

图 3-1 所示对话框中看起来有很多项目需要设置,其实除了项目名称必须输入外,其他的完全可以采用默认值。选项设置完成后,单击"下一步"按钮,进入如图 3-2 所示的"Java 设置"对话框,在这里可以对源文件和类文件的保存文件夹进行进一步设置。确认无误后,

单击“完成”按钮创建项目。项目创建完成后，返回到如图 3-3 所示的 Eclipse 主界面。



图 3-1　“新建 Java 项目”对话框

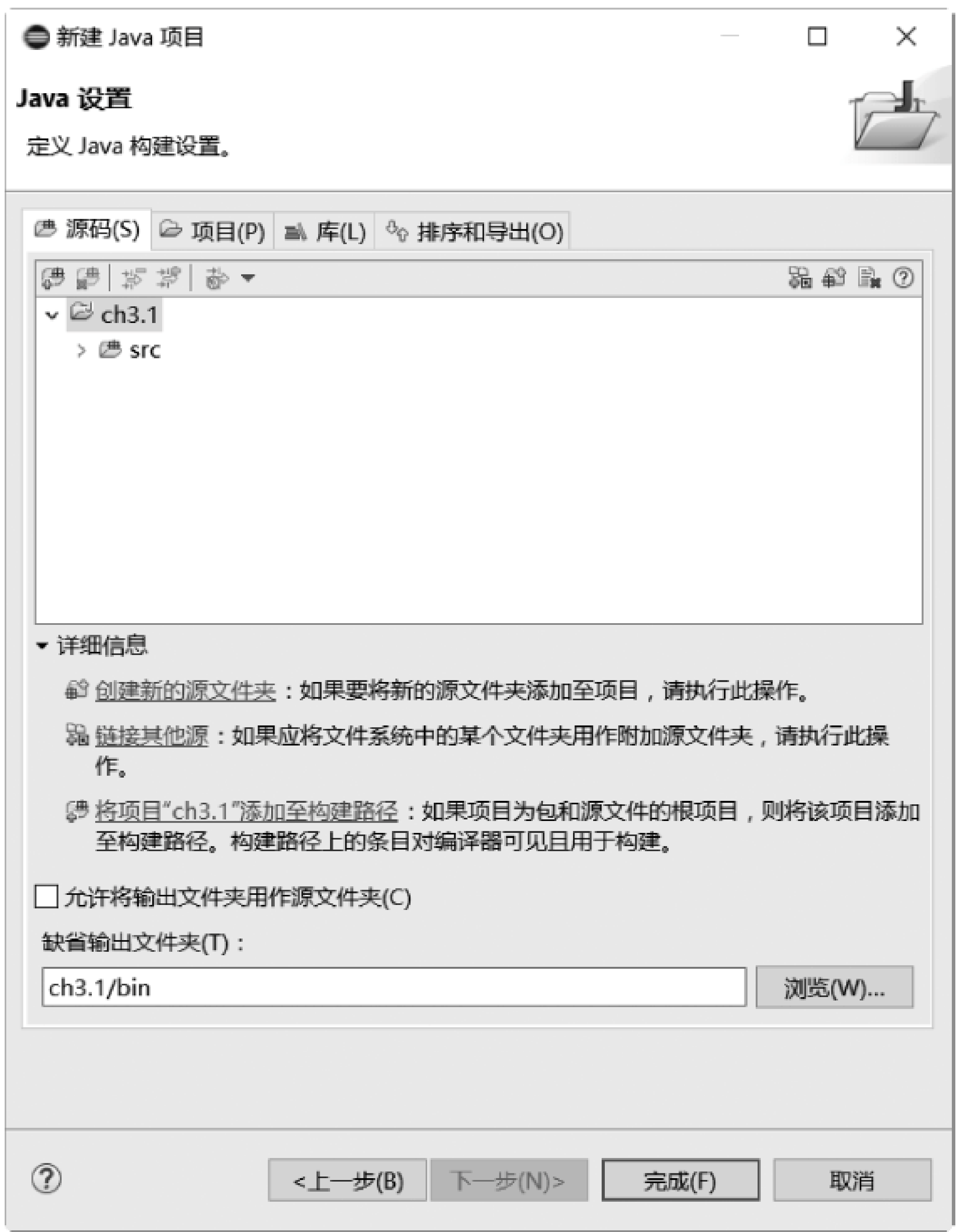


图 3-2　“Java 设置”对话框

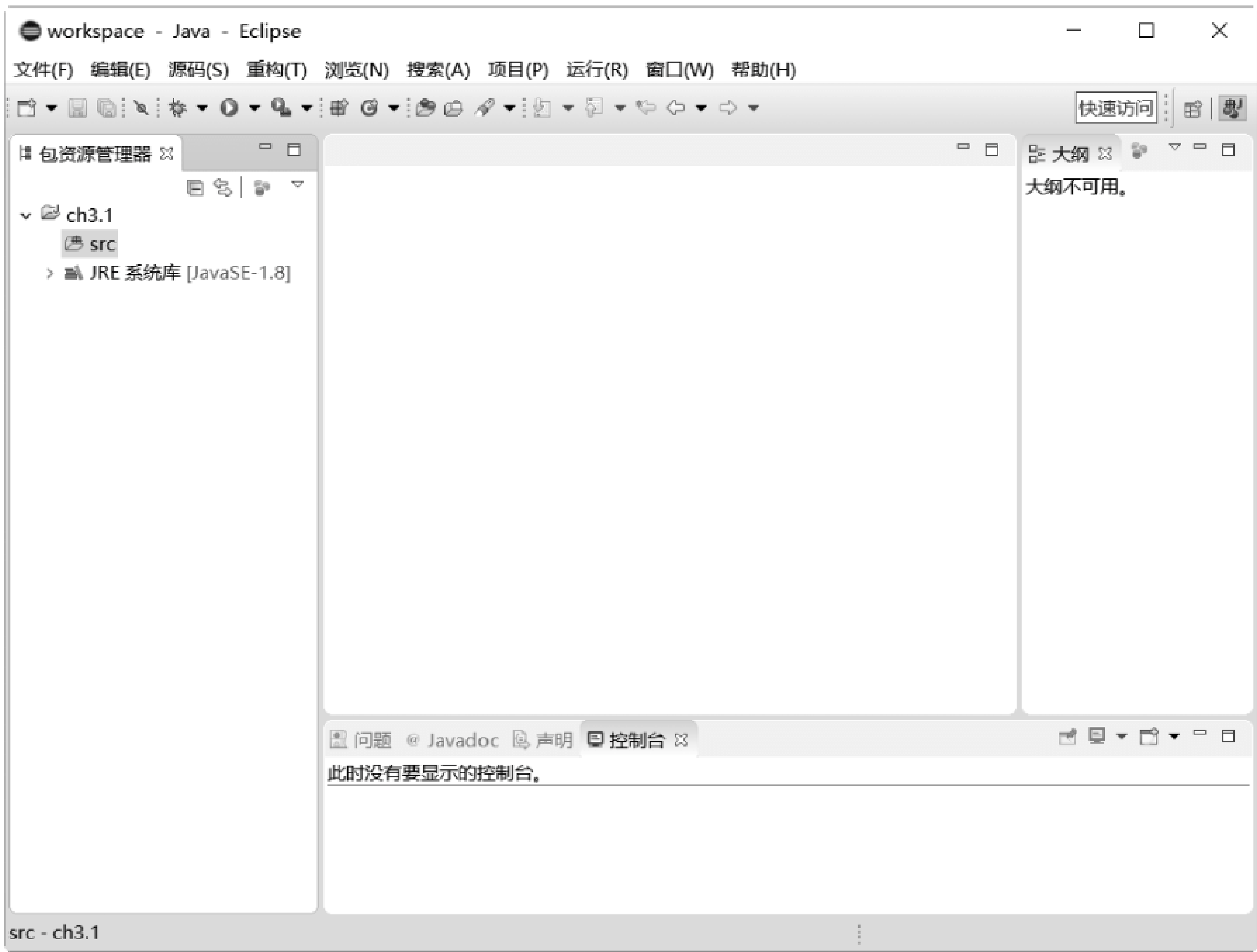


图 3-3　项目创建完成

3.1.2　创建类

项目创建完成后，需要创建一个类执行控制台输出操作。选择刚刚创建的项目，然后选择“文件”→“新建”→“类”命令，打开新建类对话框，在对话框中输入如图 3-4 所示的内容。



图 3-4　创建类对话框

下面简要说明图 3-4 中的各个选项。

- 源文件夹：由于创建项目时指定了源文件夹，这里使用默认值即可。
- 包：是类所在的包，包名一般是公司域名的倒置，可以没有。
- 名称：是类的名称。
- 修饰符：是类前面的修饰符，这些修饰符含义目前先不解释，选择“公用”即可。
- 超类：即父类，这里可以指定该类的父类。
- 接口：指定该类实现哪些接口。
- 想要创建哪些方法存根：就是在代码中创建这些方法，本例中需要选中第一个方法（main 方法），这个 main 方法是程序的入口。
- 添加注释：这里可以设置代码是否生成注释，也可以修改注释模板。

在图 3-4 所示的对话框中输入完成后，单击“完成”按钮就创建了一个 Java 类，如图 3-5 所示，在包资源管理器中可以看到刚才创建的源文件。

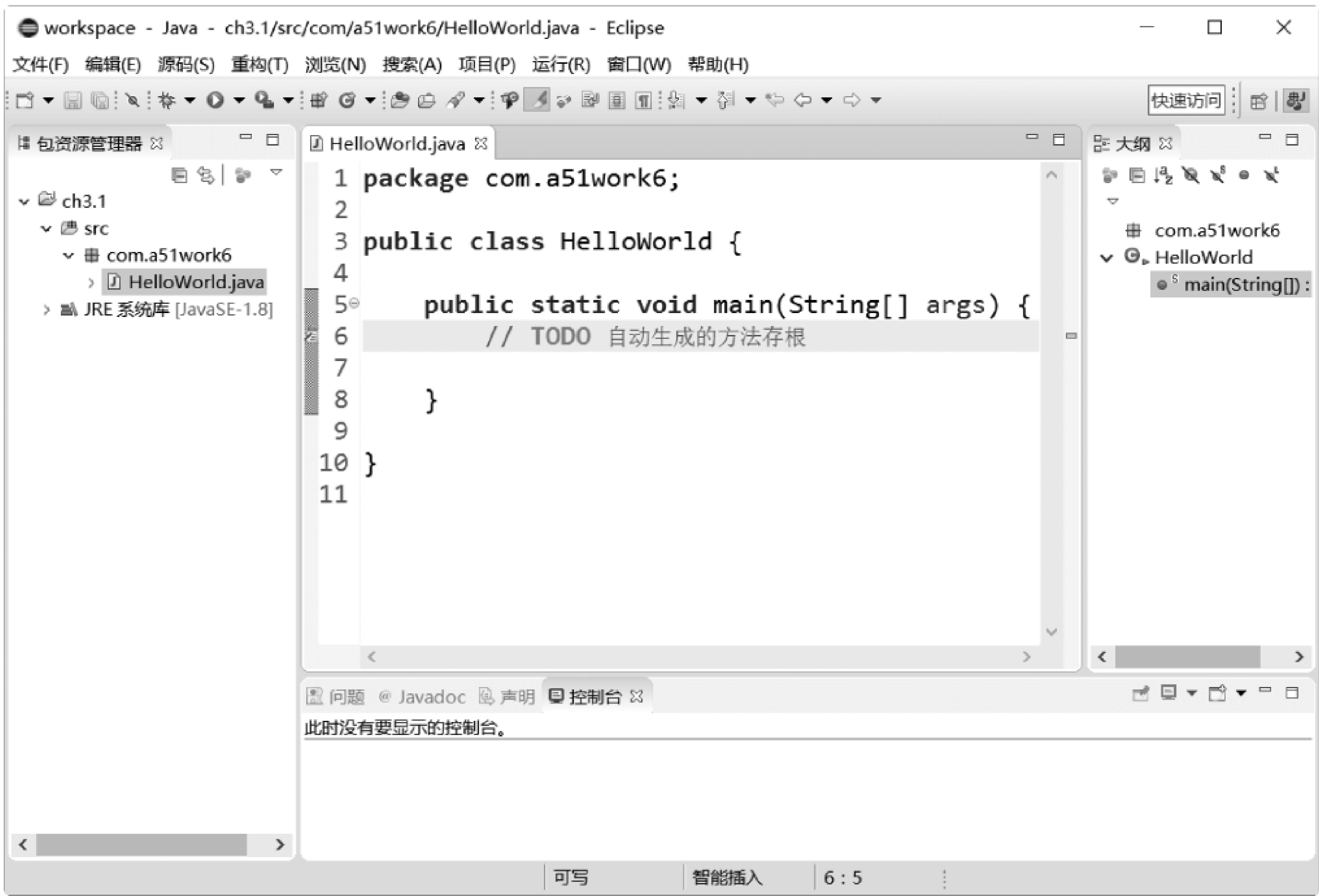


图 3-5 创建类完成

3.1.3 运行程序

修改刚刚生成的 HelloWorld.java 源文件，在 main 方法中添加输出语句。修改完成后代码如下：

```
package com.a51work6;

public class HelloWorld {
```

```
    public static void main(String[ ] args) {          ①
     System.out.print("Hello World.");                 ②
     }

}
```

代码第①行中的“public static void main(String[] args)”方法是一个应用程序的入口，也表明了HelloWorld是一个Java应用程序(Java Application)，可以独立运行。代码第②行的“System.out.print("Hello World.")”语句是输出"Hello World."字符串到控制台。

提示 在Java SE平台有两种可以独立运行的程序，即Java Application(Java应用程序)和Java Applet(Java小应用程序)。Java应用程序具有public static void main(String[] args)，上述HelloWorld就是这种类型。Java小应用程序主要是嵌入到网页中运行的，Java小应用程序是一种淘汰的技术，这里不再介绍。

程序编写完毕就可以运行了。如果是第一次运行，则需要选择运行方法，具体步骤：选中文件，选择“运行”→“运行方法”→“Java应用程序”命令，这样就会运行HelloWorld程序。如果已经运行过一次，就不需要这么麻烦了，直接单击工具栏中的“运行”按钮，或选择“运行”→“运行”命令，或使用快捷键Ctrl+F11，就可以运行上次的程序。运行结果如图3-6所示，则“Hello World.”字符串显示到下面的控制台。

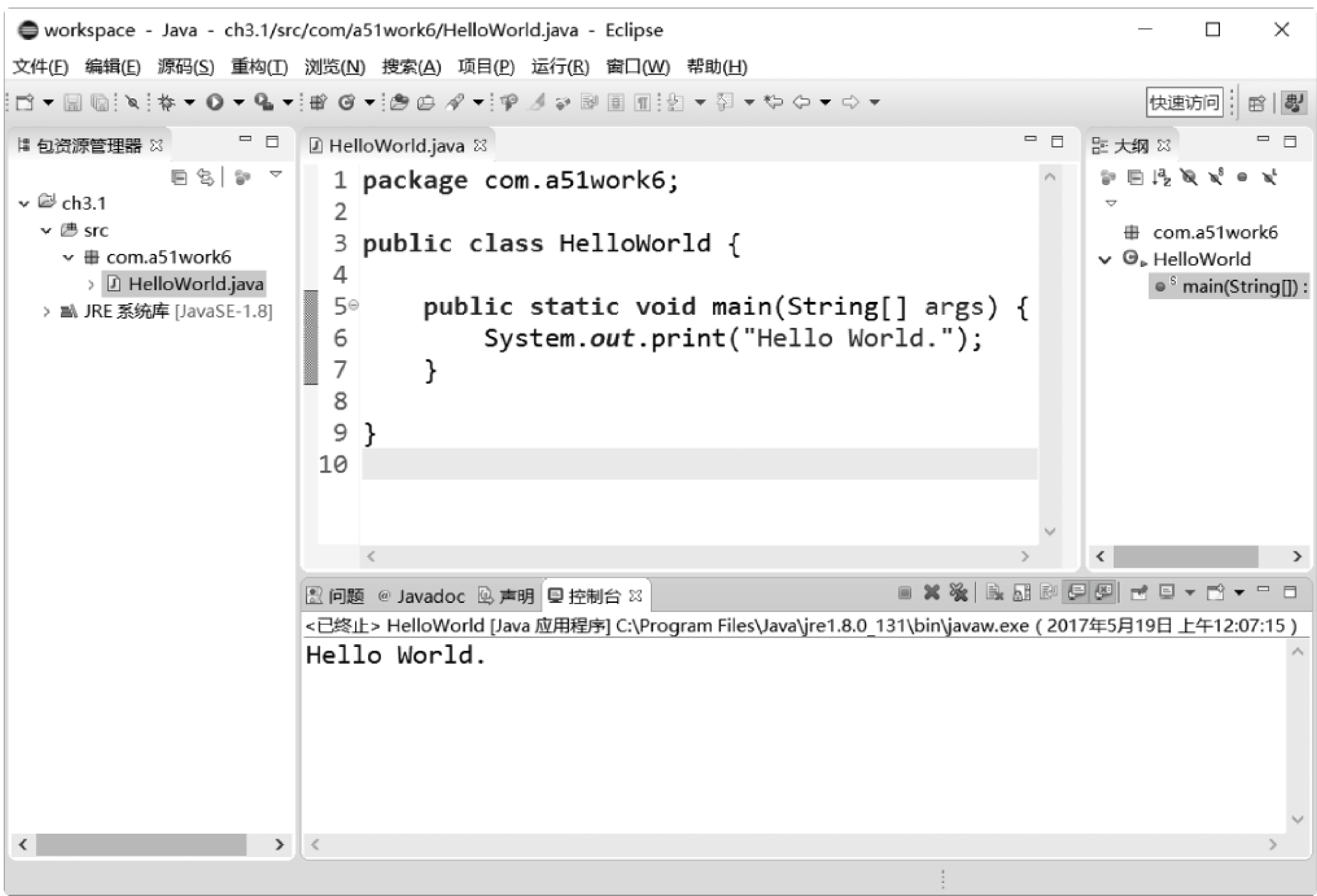


图3-6 运行结果

3.2　文本编辑工具＋JDK 实现

如果不想使用 IDE 工具(建议初学者通过这种方式学习 Java)，那么文本编辑工具＋JDK 对于初学者而言也是一个不错的选择，这种方式可以使初学者了解到 Java 程序的编译和运行过程，通过在编辑器中输入所有代码，可以帮助自己熟悉常用类和方法。

注意　在 2.3.3 节介绍过 EditPlus 与 JDK 集成过程，2.3.3 节集成方式有一个弊端是不能执行带有包的 Java 应用程序。

3.2.1　编写源代码文件

首先使用任何文本编辑工具创建一个文件，然后将文件保存为 HelloWorld.java，接着在 HelloWorld.java 文件中编写如下代码：

```
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        System.out.print("Hello World.");
    }

}
```

在 Java 的一个源程序文件中可以定义多个类，如下代码定义了 3 个类：HelloWorld、A 和 B：

```
//HelloWorld.java 源文件
package com.a51work6;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

class A {

}

class B {

}
```

注意　一个源程序文件包含多个类时，需要注意如下问题。

(1) 只能有一个类声明为公有(public)的。

(2) 文件命名必须与公有类名完全一致，包括字母大小写。

(3) public static void main(String[] args)只能定义在公有类中。

3.2.2 编译程序

编译程序需要在命令行中使用 JDK 的 javac 指令编写，参考 2.1.2 节打开命令行，如图 3-7 所示，通过 cd 命令进入到源文件所在的目录，然后执行 javac 指令。如果没有错误提示，则说明编译成功。编译成功时会在当前目录下生成类文件，如图 3-8 所示生成了 3 个类文件，这是因为 HelloWorld.java 源文件中定义了 3 个类。

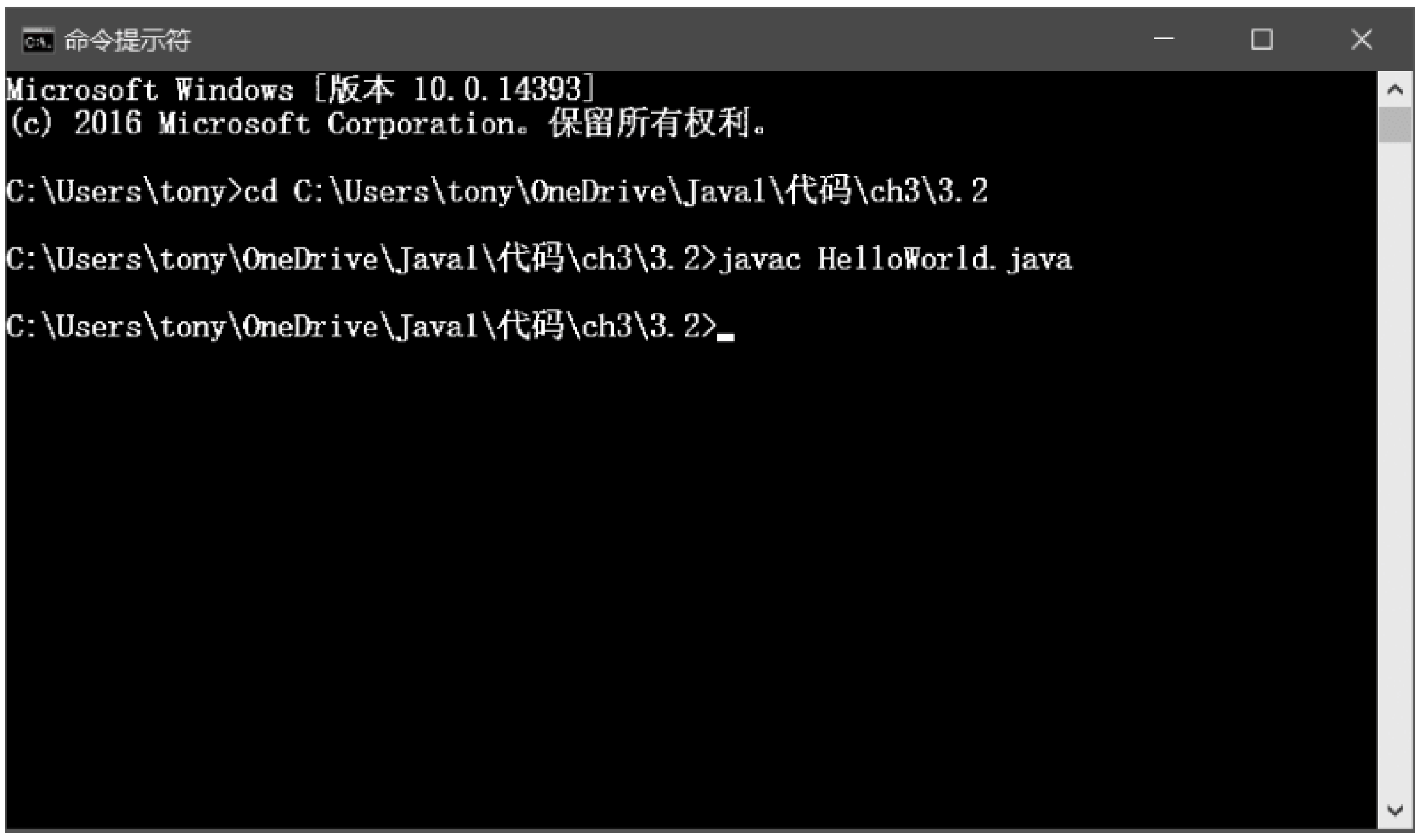


图 3-7 编译源文件



图 3-8 编译成功

上述编译过程虽然成功了，但是运行时会出现问题，这是由于 HelloWorld.java 源文件中定义了包 com.a51work6，编译应该使用-d 参数。编译指令如图 3-9 所示。

编译指令 javac 中的-d 参数是指定类文件生成位置，-d 后面跟着的是一个目录的路径，本例中使用点(.)表示当前目录。编译成功之后的目录结果如下。

```
当前目录
│ HelloWorld.java
└─com
    └─a51work6
          A.class
          B.class
          HelloWorld.class
```

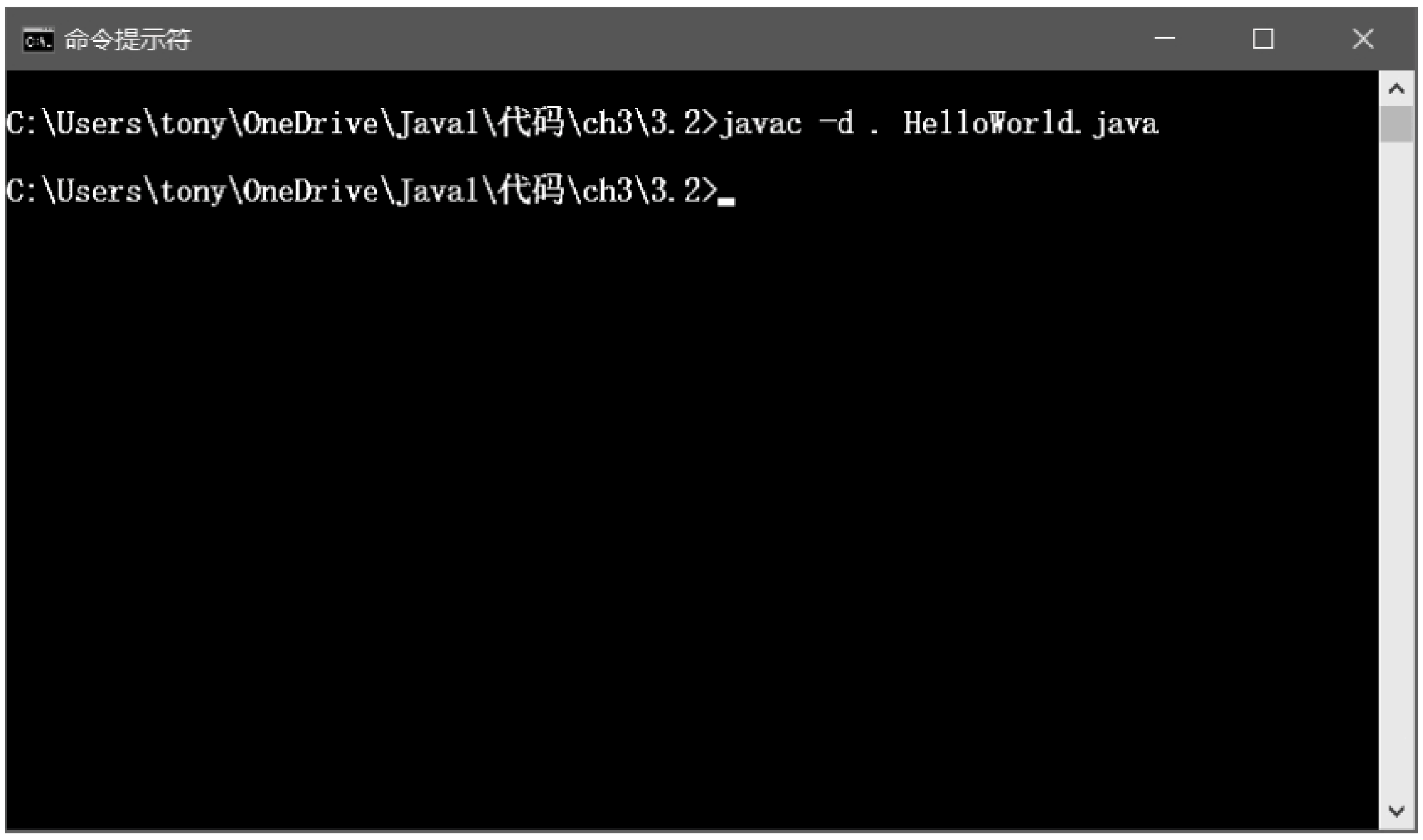


图 3-9　编译有包的源文件

其中 com 是目录，它是当前目录的子目录；a51work6 也是目录，它是 com 的子目录。可见包 com.a51work6 会生成 com\a51work6 的目录结构。

3.2.3　运行程序

编译成功之后就可以运行了。执行类文件需要在命令行中使用 JDK 的 java 指令，参考 2.1.2 节打开命令行，如图 3-10 所示，通过 cd 命令进入到源文件所在的目录，然后执行 java -classpath .；c:\com.a51work6.HelloWorld 指令，执行成功在命令行窗口输出"Hello World!"字符串。

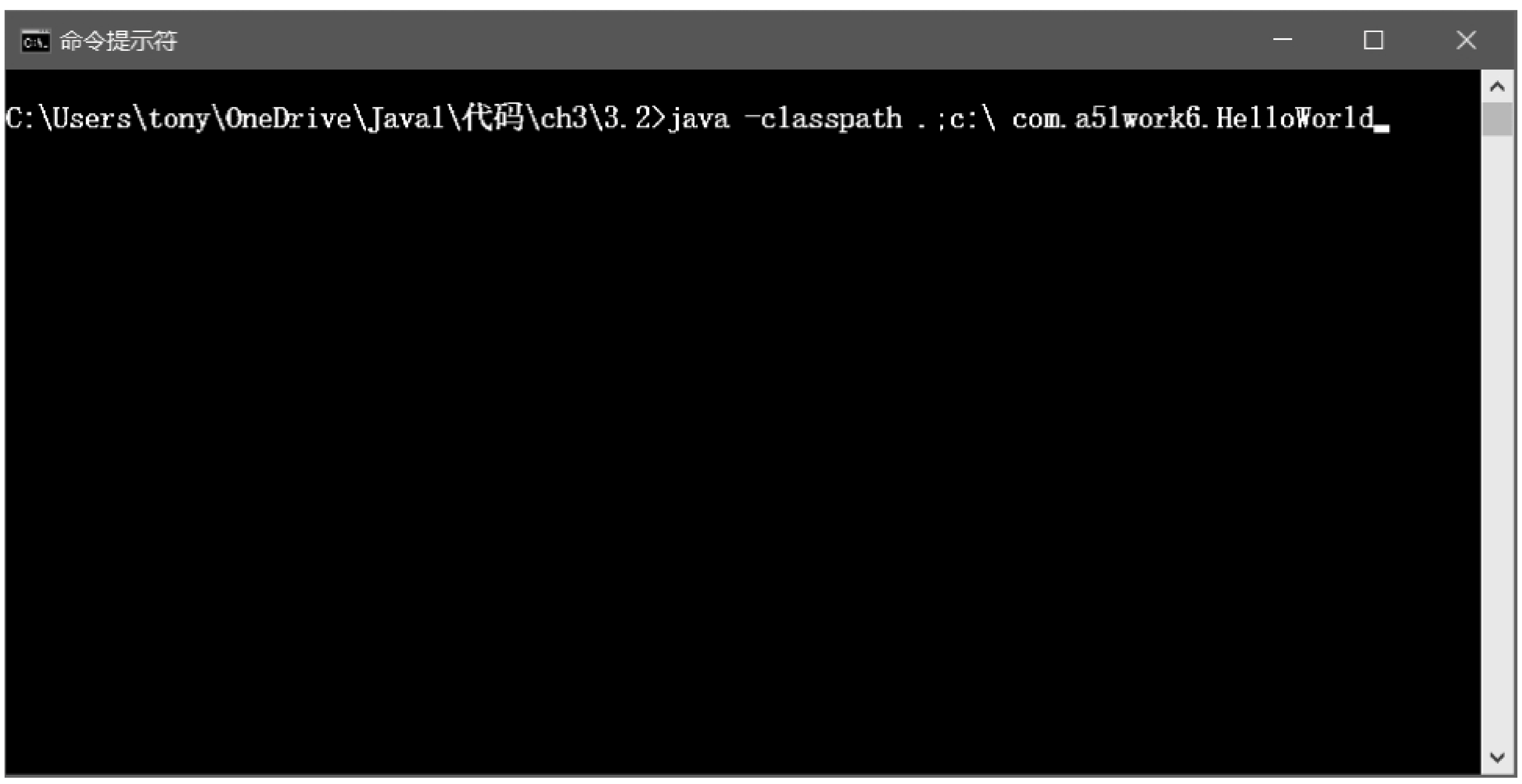


图 3-10　运行类文件

注意　java 和 javac 指令都可以带有-classpath（缩写-cp），它用来指定类路径，即搜索类的路径，类似于操作系统中的 path，路径之间用分号分隔，其中点（.）表示当前路径。

就本例而言，运行 Java 程序 HelloWorld 所需要的全部类都在当前路径下，因此只需要设置-classpath 就可以了，或者省略（当前路径不用指定）。

3.3 代码解释

经过前面的介绍，读者应该能够照猫画虎，自己动手做一个 Java 应用程序了。但可能还是对其中的一些代码不甚了解，下面来详细解释一下 HelloWorld 示例中的代码：

```
//包定义
package com.a51work6;                                    ①

//类定义
public class HelloWorld {                                ②

    //定义静态 main 方法
    public static void main(String[] args) {             ③
        System.out.print("Hello World.");                ④
    }

}
```

代码第①行是定义类所在的包，package 是关键字，com.a51work6 是包名，包是一个命名空间，可以防止命名冲突问题。关于包的概念将在后面章节详细介绍。

代码第②行是定义类，public 修饰符用于声明类是公有的，class 是定义类关键字，HelloWorld 是自定义的类名，后面跟着的"{…}"是类体，类体中会有成员变量和方法，也会有一些静态变量和方法。

代码第③行是定义静态 main 方法，而作为一个 Java 应用程序，类中必须包含静态 main 方法，程序执行是从 main 方法开始的。main 方法中除参数名 args 可以自定义外，其他必须严格遵守如下两种格式：

```
public static void main(String args[])
public static void main(String[] args)
```

这两种格式本质上就是一种，String args[]和 String[] args 都是声明 String 数组。另外，args 参数是程序运行时通过控制台向应用程序传递字符串参数。

代码第④行"System.out.print("Hello World.");"语句是通过 Java 输出流（PrintStream）对象 System.out 打印"Hello World."字符串，System.out 是标准输出流对象，它默认输出到控制台。输出流（PrintStream）中常用打印方法如下。

- print(String s)：打印字符串不换行，有多个重载方法，可以打印任何类型数据。
- println(String x)：打印字符串换行，有多个重载方法，可以打印任何类型数据。
- printf(String format, Object … args)：使用指定输出格式，打印任何长度的数据，但不换行。

修改 HelloWorld.java 示例代码如下：

```
public class HelloWorld {
 public static void main(String[] args) {

    //通过 print 打印第一个控制台参数
    System.out.print(args[0]);                        ①
    //通过 println 打印第二个控制台参数
    System.out.println(args[1]);                      ②
    //通过 printf 打印第三个控制台参数,%s 表示格式化字符串
    System.out.printf("%s", args[2]);                 ③
    System.out.println();

    int i = 123;
    //%d 表示格式化整数
    System.out.printf("%d\n", i);                     ④

    double d = 123.456;
    //%f 表示格式化浮点数
    System.out.printf("%f%n", d);                     ⑤
    System.out.printf("%5.2f", d);                    ⑥

 }
}
```

编译 HelloWorld.java 源代码后，如图 3-11 所示，其中的 java 命令行后面的 HelloWorld 是要运行的类文件，Tony Hello World. 是参数，多个参数用空格分隔。

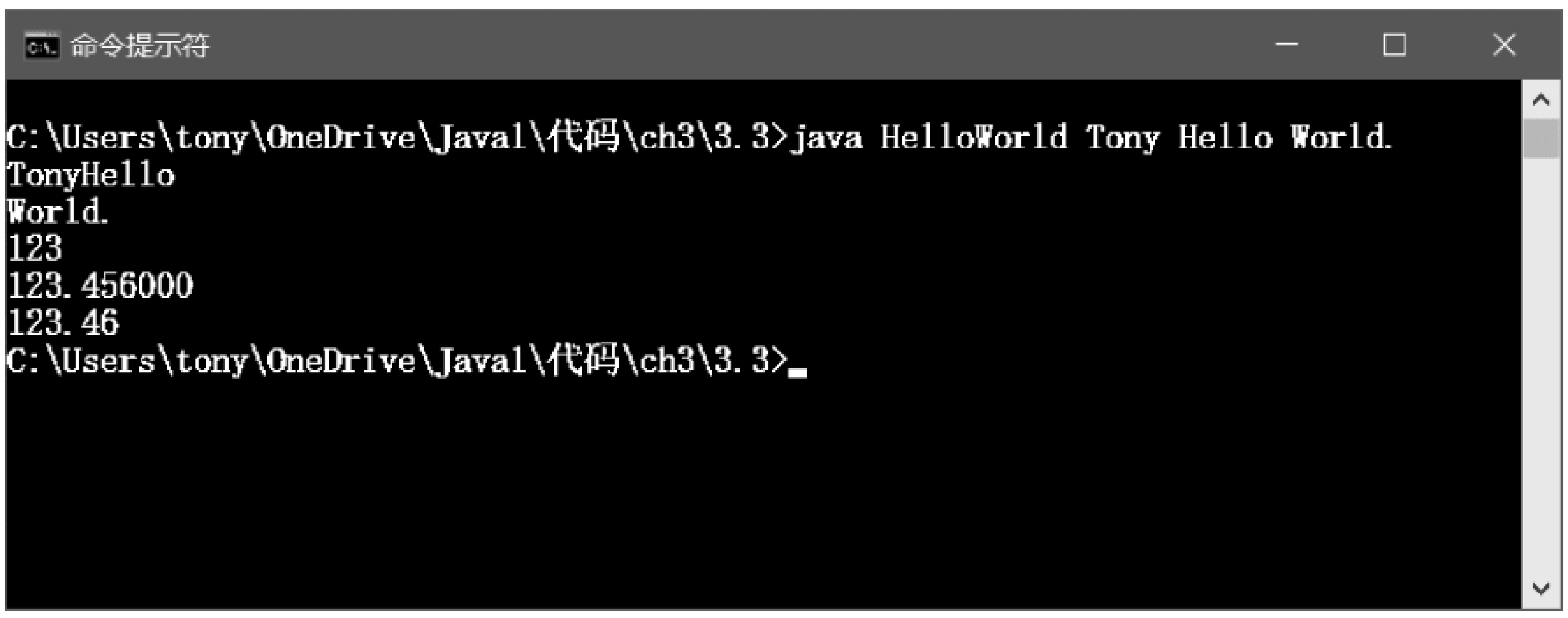


图 3-11　在命令行中运行程序

上述代码第①行使用 print 方法打印第一个参数 args[0]，注意该方法是打印完成后不换行，从输出结果中可见第一个参数 Tony 和第二个参数 Hello 连在一起了。代码第②行使用 println 方法打印第二个参数 args[1]，从输出结果中可见第二个参数 Hello 后面是有换行的。

代码第③～⑥行都是使用 printf 方法打印，注意 printf 方法后面是没有换行的，想在后面换行可以通过 System.out.println()语句实现，或在打印的字符串后面添加换行符号(\n 或%n)，见代码第④行和第⑤行。代码第⑥行中的%5.2f 也表示格式化浮点数，5 表示总输出的长度，2 表示保留的小数位。

本章小结

本章通过一个 HelloWorld 示例，介绍使用 Eclipse 和使用文本工具＋JDK 实现该示例的具体过程。掌握 Eclipse 使用非常重要，但是使用文本工具＋JDK 对于初学者也很有帮助。最后详细解释了 HelloWorld 示例。

3.4 同步练习

1. 使用 Eclipse 工具编写并运行 Java 程序，使其在控制台输出字符串"世界，你好！"。
2. main()方法的返回类型是(　　)。

A. int　　　　B. void

C. boolean　　　　D. static

第 4 章 Java 语法基础

CHAPTER 4

本章主要介绍 Java 的一些基本语法，其中包括标识符、关键字、保留字、常量、变量等内容。

4.1 标识符、关键字和保留字

任何一种计算机语言都离不开标识符和关键字，下面将详细介绍 Java 标识符、关键字和保留字。

4.1.1 标识符

标识符就是变量、常量、方法、枚举、类、接口等由程序员指定的名字。构成标识符的字母均有一定的规范要求，Java 语言中标识符的命名规则如下。

(1) 区分大小写。Myname 与 myname 是两个不同的标识符。

(2) 首字符。可以是下画线(_)或美元符($)或字母，但不能是数字。

(3) 除首字符外其他字符。可以是下画线(_)、美元符($)、字母和数字。

(4) 关键字不能作为标识符。

例如，身高、identifier、userName、User_Name、$Name、_sys_val 等为合法的标识符，注意中文“身高”命名的变量是合法的；而 2mail、room# 和 class 为非法的标识符，注意 # 是非法字符，而 class 是关键字。

注意 Java 语言中字母采用的是双字节 Unicode 编码[①]。Unicode 叫作统一编码制，它包含了亚洲文字编码，如中文、日文、韩文等字符。

4.1.2 关键字

关键字是类似于标识符的保留字符序列，是由语言本身定义好的，不能挪作他用。Java 语言中有 50 个关键字，如表 4-1 所示。

表 4-1 Java 关键字

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else

① Unicode 是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。

续表

enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	strictfp	short	static	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

关键字很多，这里不再一一介绍了，但是需要记住一点的是，Java 中的关键字全部是小写字母。

4.1.3 保留字

Java 中有一些字符序列既不能当作标识符使用，也不是关键字，不能在程序中使用，这些字符序列称为保留字。Java 语言中的保留字只有两个，即 goto 和 const。

(1) goto：在其他语言中叫作"无限跳转"语句，在 Java 语言中不再使用 goto 语句，因为"无限跳转"语句会破坏程序结构。在 Java 语言中，goto 的替换语句可以通过 break、continue 和 return 实现"有限跳转"。

(2) const：在其他语言中是声明常量关键字，在 Java 语言中声明常量使用 public static final 方式。

4.2 Java 分隔符

在 Java 源代码中，有一些字符被用作分隔，称为分隔符。分隔符主要有分号(;)、左右大括号({})和空白。

1. 分号

分号是 Java 语言中最常用的分隔符，它表示一条语句的结束。下面代码：

```
int totals = 1 + 2 + 3 + 4;
```

等价于

```
int totals = 1 + 2
        + 3 + 4;
```

2. 大括号

在 Java 语言中，以左右大括号({})括起来的语句集合称为语句块(block)或复合语句，语句块中可以有 0～n 条语句。在定义类或方法时，语句块也被用作分隔类体或方法体。语句块也可以嵌套，且嵌套层次没有限制。示例代码如下：

```
public class HelloWorld {

    public static void main(String args[]) {
```

```
        int m = 5;
        if(m < 10) {
            System.out.println("< 10");
        }

    }
}
```

3. 空白

在 Java 源代码中元素之间允许有空白，空白的数量不限。空白包括空格、制表符(Tab 键输入)和换行符(Enter 键输入)，适当的空白可以改善对源代码的可读性。下列几段代码是等价的：

```
if (m < 10) {
    System.out.println("< 10"); }
```

等价于

```
if (m < 10)
    {
    System.out.println("< 10");
}
```

等价于

```
if (m < 10) {
    System.out.println("< 10");
}
```

4.3 变量

变量是构成表达式的重要部分，变量所代表的内容是可以被修改的。变量包括变量名和变量值，变量的声明格式为：

```
数据类型  变量名  [ = 初始值];
```

变量名要遵守标识符命名规范，即在相关的作用域中不能有重复的变量名。变量作用域是变量的使用范围，在此范围内变量可以使用，超过作用域，变量内容则被释放，根据作用域不同分为成员变量和局部变量。示例代码如下：

```
public class HelloWorld {

    //声明 int 型成员变量
    int y;                                                   ①

    public static void main(String[ ] args) {
```

```
        //声明 int 型局部变量
        int x;                                                          ②
        //声明 float 型变量并赋值
        float f = 4.5f;                                                 ③

        //x = 10;
        System.out.println("x = " + x);        //编译错误,局部变量 x 未初始化  ④
        System.out.println("f = " + f);

        if (f < 10) {
            //声明 int 型局部变量
            int m = 5;                                                  ⑤

        }
        System.out.println(m);                  //编译错误                ⑥
    }
}
```

上述代码中,代码第①行是声明的成员变量 y,成员变量是在类体中,而在方法之外,作用域是整个类,如果没有初始赋值,系统会为它分配一个默认值,每一种数据类型都有默认值,int 类型默认值是 0。

代码第②、③、⑤行都是声明局部变量,局部变量是在方法或 if、for 和 while 等代码块中声明的变量,第②和③行声明局部变量作用域是整个方法,第⑤行声明的 m 变量作用域是当前的 if 语句。

另外,代码第④行和第⑥行会有编译错误提示,这是因为第④行中 x 使用之前没有被初始化,与成员变量不同,局部变量在使用之前必须显式地初始化。代码第③行是在声明的同时初始化了。代码第⑥行的错误是因为 m 变量超出了作用域。

4.4 常量

常量事实上是那些内容不能被修改的变量。常量与变量类似,也需要初始化,即在声明常量的同时要赋予一个初始值。常量一旦初始化就不可以被修改。它的声明格式为:

```
final 数据类型 变量名 = 初始值;
```

final 关键字表示最终的,它可以修改很多元素,修饰变量就变成了常量。示例代码如下。

```
public class HelloWorld {

    //声明静态常量,替代保留字 const
    public static final double PI = 3.14;                  ①

    //声明成员常量
    final int y = 10;                                      ②

    public static void main(String[] args) {
```

```
        //声明局部常量
        final double x = 3.3;                                    ③
    }
}
```

事实上常量有 3 种类型：静态常量、成员常量和局部常量。代码第①行声明的是静态常量，在 final 之前用 public static 修饰，用来替代保留字 const。public static 修饰的常量作用域是全局的，不需要创建对象就可以访问它，在类外部的访问形式为：HelloWorld. PI，这种常量在编程中使用很多。

代码第②行声明的是成员常量，作用域类似于成员变量，但不能修改。代码第③行声明的是局部常量，作用域类似于局部变量，但不能修改。

本章小结

本章主要介绍了 Java 语言中最基本的语法，首先介绍了标识符、关键字和保留字，读者需要掌握标识符构成，了解 Java 关键字和保留字。接着介绍了 Java 中的分隔符，最后介绍了变量和常量，读者需要掌握变量种类和作用域，以及常量的声明。

4.5 同步练习

1. 下面哪些是 Java 的保留字？(　　)

A. if　　B. then
C. goto　　D. while
E. case

2. 下面哪些是 Java 的合法标识符？(　　)

A. 2variable　　B. variable2
C. _whatavariable　　D. _3_
E. $ anothervar　　F. # myvar

3. 判断对错。

(1) 在 Java 语言中，一行代码表示一条语句。语句结束可以加分号，也可以省略分号。(　　)

(2) Java 语言中的保留字只有两个，即 goto 和 const。可以使用保留字声明变量。(　　)

第 5 章 数据类型

CHAPTER 5

在声明变量或常量时会用到数据类型，在前面已经用到一些数据类型，如 int、double 和 String 等。Java 语言的数据类型分为基本数据类型和引用数据类型。

5.1 基本数据类型

基本数据类型表示简单的数据，基本数据类型分为 4 大类，共 8 种数据类型。

(1) 整数类型：byte、short、int 和 long。

(2) 浮点类型：float 和 double。

(3) 字符类型：char。

(4) 布尔类型：boolean。

基本数据类型如图 5-1 所示，其中整数类型、浮点类型和字符类型都属于数值类型，它们之间可以互相转换。

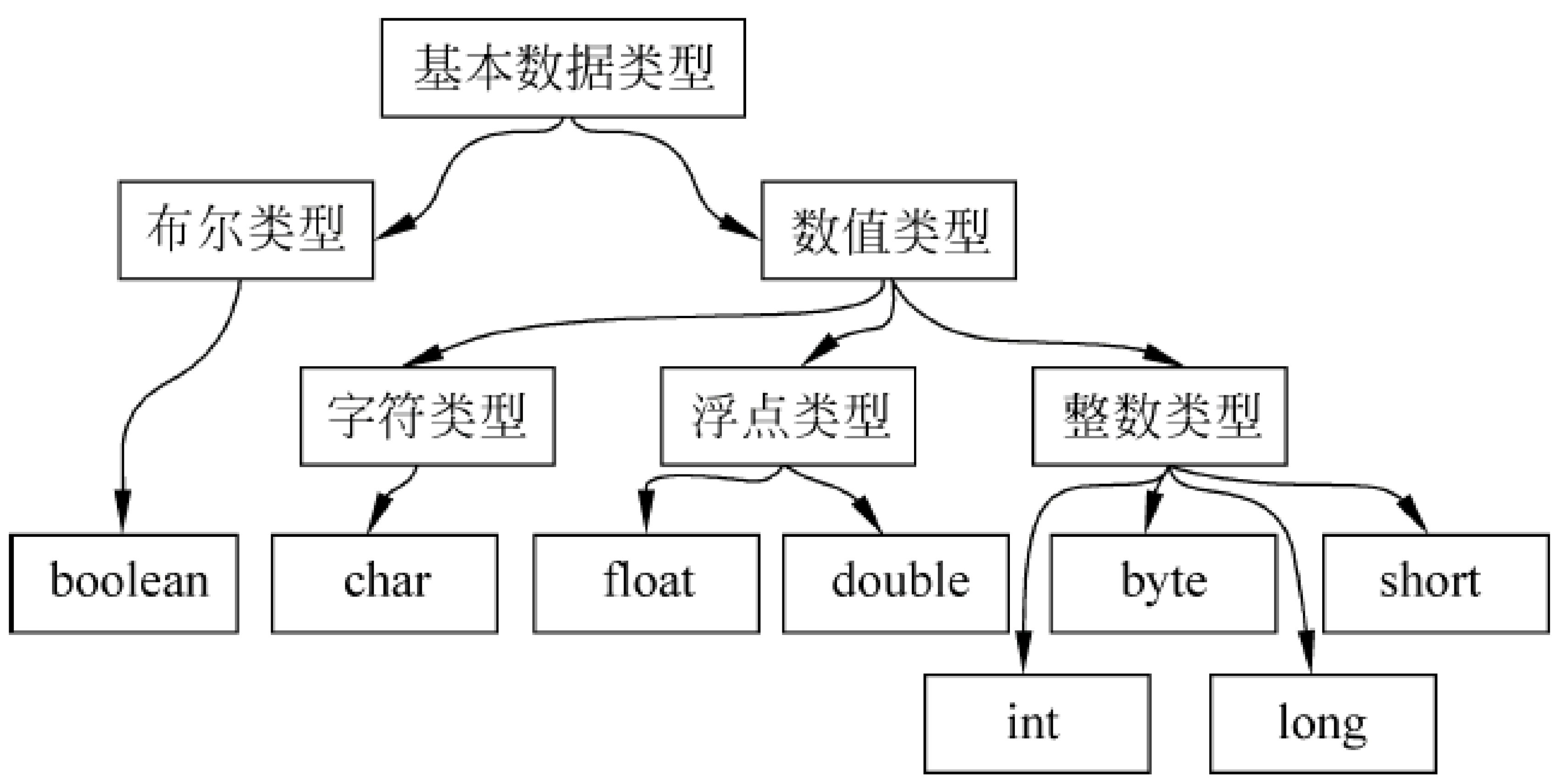


图 5-1 基本数据类型

5.2 整型类型

从图 5-1 中可见，Java 中整数类型包括 byte、short、int 和 long，它们之间的区别仅仅是宽度和范围的不同。Java 中整数都有符号，与 C 语言不同，没有无符号的整数类型。

Java 的数据类型是跨平台的（与平台无关），无论计算机是 32 位的还是 64 位的，byte 类型整数都是 1 字节（8 位）。这些整数类型的宽度和范围如表 5-1 所示。

表 5-1　整数类型

整数类型	宽　　度	取值范围
byte	1 字节(8 位)	$-128\sim127$
short	2 字节(16 位)	$-2^{15}\sim2^{15}-1$
int	4 字节(32 位)	$-2^{31}\sim2^{31}-1$
long	8 字节(64 位)	$-2^{63}\sim2^{63}-1$

Java 语言的整数类型默认是 int 类型,例如 16 表示为 int 类型常量,而不是 short 或 byte,更不是 long,long 类型需要在数值后面加 l(小写英文字母)或 L(大写英文字母)。示例代码如下:

```
public class HelloWorld {

    public static void main(String[] args) {
        //声明整数变量
        //输出一个默认整数常量
        System.out.println("默认整数常量 = " + 16);                ①
        byte a = 16;                                              ②
        short b = 16;                                             ③
        int c = 16;                                               ④
        long d = 16L;                                             ⑤
        long e = 16l;                                             ⑥

        System.out.println("byte 整数    = " + a);
        System.out.println("short 整数   = " + b);
        System.out.println("int 整数     = " + c);
        System.out.println("long 整数    = " + d);
        System.out.println("long 整数    = " + e);

    }
}
```

上述代码多次用到了 16 整数,但它们是有所区别的。其中,代码第①行的 16 是默认整数类型,即 int 类型常量;代码第②行的 16 是 byte 整数类型;代码第③行的 16 是 short 类型;代码第④行的 16 是 int 类型;代码第⑤行的 16 后加了 L,这说明是 long 类型整数;代码第⑥行的 16 后加了 l(小写英文字母 l),这也是 long 类型整数。

提示　在程序代码中,尽量不用小写英文字母 l,因为它容易与数字 1 混淆,特别是在 Java 中表示 long 类型整数时很少使用小写英文字母 l,而是使用大写的英文字母 L。例如,16L 要比 16l 可读性更好。

5.3　浮点类型

浮点类型主要用来储存小数数值,也可以用来储存范围较大的整数。它分为浮点数(float)和双精度浮点数(double)两种,双精度浮点数所使用的内存空间比浮点数多,可表示的数值范围比较大,精确度也比较高。浮点类型的说明如表 5-2 所示。

表 5-2 浮点类型

浮点类型	宽　度
float	4 字节(32 位)
double	8 字节(64 位)

Java 语言的浮点类型默认是 double 类型,例如 0.0 表示 double 类型常量,而不是 float 类型。如果想要表示 float 类型,则需要在数值后面加 f 或 F。示例代码如下:

```
public class HelloWorld {

    public static void main(String[ ] args) {
        //声明浮点数
        //输出一个默认浮点常量
        System.out.println("默认浮点常量 = " + 360.66);          ①
        float myMoney = 360.66f;                                 ②
        double yourMoney = 360.66;                               ③
        final double PI = 3.14159d;                              ④

        System.out.println("float 浮点数 = " + myMoney);
        System.out.println("double 浮点数 = " + yourMoney);
        System.out.println("PI = " + PI);

    }
}
```

上述代码第①行的 360.66 是默认浮点类型 double;代码第②行的 360.66f 是 float 浮点类型,float 浮点类型常量表示时,数值后面需要加 f 或 F;代码第③行的 360.66 表示是 double 浮点类型,事实上 double 浮点数值后面也可以加字母 d 或 D,以表示是 double 浮点数;代码第④行是声明一个 double 类型常量,数值后面加了 d 字母。

5.4 数值表示方式

整数类型和浮点类型都表示数值类型,那么在给这些类型的变量或常量赋值时,应该如何表示这些数值呢?下面介绍数值的进制表示和指数表示方式。

5.4.1 进制数值表示

如果为一个整数变量赋值,使用二进制数、八进制数和十六进制数表示,它们的表示方式分别如下。

- 二进制数:以 0b 或 0B 为前缀,注意 0 是阿拉伯数字,不要误认为是英文字母 o。
- 八进制数:以 0 为前缀,注意 0 是阿拉伯数字。
- 十六进制数:以 0x 或 0X 为前缀,注意 0 是阿拉伯数字。

例如,下面几条语句都是表示 int 整数 28。

```
int decimalInt = 28;
int binaryInt1 = 0b11100;
```

```
int binaryInt2 = 0B11100;
int octalInt = 034;
int hexadecimalInt1 = 0x1C;
int hexadecimalInt2 = 0X1C;
```

5.4.2　指数表示

进行数学计算时往往会用到指数表示的数值。如果采用十进制表示指数，则需要使用大写 E 或小写的 e 表示幂，e2 表示 10^2。

采用十进制指数表示的浮点数示例如下：

```
double myMoney = 3.36e2;
double interestRate = 1.56e-2;
```

其中，3.36e2 表示的是 3.36×10^2，1.56e−2 表示的是 1.56×10^{-2}。

5.5　字符类型

字符类型表示单个字符，Java 中 char 声明字符类型，Java 中的字符常量必须是用单引号括起来的单个字符，如下所示：

```
char c = 'A';
```

Java 字符采用双字节 Unicode 编码，占 2 字节（16 位），因而可用十六进制（无符号的）编码形式表示，它们的表现形式是\un，其中 n 为 16 位十六进制数，所以 'A' 字符也可以用 Unicode 编码 '\u0041' 表示。如果对字符编码感兴趣，可以到维基百科（https://zh.wikipedia.org/wiki/Unicode 字符列表）查询。

示例代码如下：

```
public class HelloWorld {

    public static void main(String[] args) {

        char c1 = 'A';
        char c2 = '\u0041';
        char c3 = '花';

        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
    }
}
```

上述代码变量 c1 和 c2 都是保存的 'A'，所以输出结果如下：

```
A
A
花
```

提示 字符类型也属于数值类型，可以与 int 等数值类型进行数学计算或进行转换。这是因为字符类型在计算机中保存的是 Unicode 编码，双字节 Unicode 的存储范围为 \u0000～\uFFFF，所以 char 类型取值为 $0 \sim 2^{16}-1$。

在 Java 中，为了表示一些特殊字符，前面要加上反斜杠(\)，这称为字符转义。常见的转义符的含义如表 5-3 所示。

表 5-3 转义符

字符表示	Unicode 编码	说明
\t	\u0009	水平制表符 Tab
\n	\u000a	换行
\r	\u000d	回车
\"	\u0022	双引号
\'	\u0027	单引号
\\	\u005c	反斜线

示例如下：

```
//在 Hello 和 World 之间插入制表符
String specialCharTab1 = "Hello\tWorld.";
//在 Hello 和 World 之间插入制表符,制表符采用 Unicode 编码\u0009 表示
String specialCharTab2 = "Hello\u0009World.";
//在 Hello 和 World 之间插入换行符
String specialCharNewLine = "Hello\nWorld.";
//在 Hello 和 World 之间插入回车符
String specialCharReturn = "Hello\rWorld.";
//在 Hello 和 World 之间插入双引号
String specialCharQuotationMark = "Hello\"World\".";
//在 Hello 和 World 之间插入单引号
String specialCharApostrophe = "Hello\'World\'.";
//在 Hello 和 World 之间插入反斜杠
String specialCharReverseSolidus = "Hello\\World.";

System.out.println("水平制表符 Tab1: " + specialCharTab1);
System.out.println("水平制表符 Tab2: " + specialCharTab2);
System.out.println("换行: " + specialCharNewLine);
System.out.println("回车: " + specialCharReturn);
System.out.println("双引号: " + specialCharQuotationMark);
System.out.println("单引号: " + specialCharApostrophe);
System.out.println("反斜杠: " + specialCharReverseSolidus);
```

输出结果如下：

```
水平制表符 Tab1: HelloWorld.
水平制表符 Tab2: HelloWorld.
换行: Hello
World.
回车: Hello
World.
双引号: Hello"World".
单引号: Hello'World'.
反斜杠: Hello\World.
```

5.6　布尔类型

在 Java 语言中声明布尔类型的关键字是 boolean，它只有两个值：true 和 false。

提示　在 C 语言中布尔类型是数值类型，它有两个取值，即 1 和 0。而在 Java 中的布尔类型取值不能用 1 和 0 替代，也不属于数值类型，不能与 int 等数值类型之间进行数学计算或类型转化。

示例代码如下：

```
boolean isMan = true;
boolean isWoman = false;
```

如果试图给它们赋值 true 和 false 之外的常量，如下所示：

```
boolean isMan = 1;
boolean isWoman = 'A';
```

则发生类型不匹配编译错误。

5.7　数值类型相互转换

学习了前面的数据类型后，大家会思考一个问题，数据类型之间是否可以转换呢？数据类型的转换情况比较复杂。基本数据类型中数值类型之间可以互相转换，布尔类型不能与它们之间进行转换。但有些不兼容类型之间，如 String（字符串）转换为 int 整数等，可以借助于一些类的方法实现。本节只讨论数值类型的互相转换。

从图 5-1 中可见，数值类型包括 byte、short、char、int、long、float 和 double，这些数值类型之间的转换有两个方向：自动类型转换和强制类型转换。

5.7.1　自动类型转换

自动类型转换就是需要类型之间转换是自动的，不需要采取其他手段，总的原则是小范围数据类型可以自动转换为大范围数据类型。类型转换顺序如图 5-2 所示，从左到右是自动的。

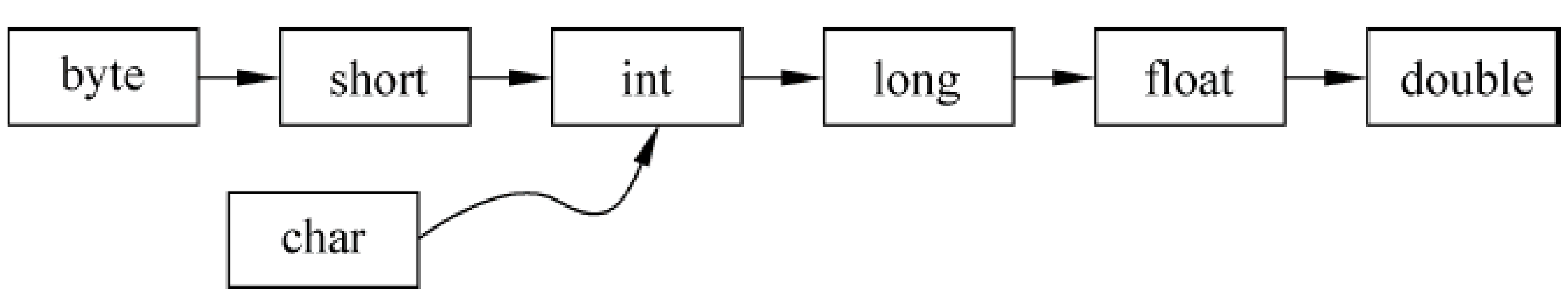


图 5-2　数据类型转换顺序

注意　如图 5-2 所示，char 类型比较特殊，char 自动转换为 int、long、float 和 double，但 byte 和 short 不能自动转换为 char，而且 char 也不能自动转换为 byte 或 short。

自动类型转换不仅发生在赋值过程中，在进行数学计算时也会发生自动类型转换，在运算中往往是先将数据类型转换为同一类型，然后再进行计算。转换规则如表 5-4 所示。

表 5-4 计算过程中自动类型的转换规则

操作数 1 类型	操作数 2 类型	转换后的类型
byte、short、char	int	int
byte、short、char、int	long	long
byte、short、char、int、long	float	float
byte、short、char、int、long、float	double	double

示例如下：

```
//声明整数变量
byte byteNum = 16;
short shortNum = 16;
int intNum = 16;
long longNum = 16L;

//byte 类型转换为 int 类型
intNum = byteNum;
//声明 char 变量
char charNum = '花';
//char 类型转换为 int 类型
intNum = charNum;

//声明浮点变量
//long 类型转换为 float 类型
float floatNum = longNum;
//float 类型转换为 double 类型
double doubleNum = floatNum;

//表达式计算后类型是 double
double result = floatNum * intNum + doubleNum / shortNum;          ①
```

上述代码第①行中表达式 floatNum * intNum + doubleNum / shortNum 进行数学计算，该表达式是由 4 个完全不同的数据类型组成，范围最大的是 double，在计算过程中它们先转换成 double，最后的结果类型是 double。

5.7.2 强制类型转换

在数值类型转换过程中，除了需要自动类型转换外，有时还需要强制类型转换，强制类型转换是在变量或常量之前加上“(目标类型)”实现。示例代码如下：

```
//int 型变量
int i = 10;
//把 int 变量 i 强制转换为 byte
byte b = (byte) i;
```

上述代码(byte) i 表达式实现强制类型转换。强制类型转换主要用于大宽度类型转换为小宽度类型情况，如把 int 转换为 byte。示例代码如下：

```
//int 型变量
int i = 10;
//把 int 变量 i 强制转换为 byte
byte b = (byte)i;
int i2 = (int)i;            ①
int i3 = (int)b;            ②
```

上述代码第①行是将 int 类型的 i 变量强制转换为 int 类型，这显然没有必要，但是语法也是允许的。代码第②行是将 byte 类型的 b 变量强制转换为 int 类型，从图 5-2 中可见这个转换是自动的，不需要强制转换。本例中这个转换没有实际意义，但有时为了提高精度需要这种转换。示例代码如下。

```
//int 型变量
int i = 10;
float c1 = i / 3;                  ①
System.out.println(c1);            ②
//把 int 变量 i 强制转换为 float
float c2 = (float)i / 3;           ③
System.out.println(c2);            ④
```

输出结果为：

```
3.0
3.3333333
```

比较上述代码输出结果发现，c1 和 c2 变量小数部分差别是比较大的，这种差别在一些金融系统中是不允许的。在代码第①行 i 除以 3 中结果是小数，但由于两个操作数都是整数 int 类型，小数部分被截掉了，结果是 3，然后再赋值给 float 类型的 c1 变量，最后 c1 保存的是 3.0。为了防止两个整数进行除法等运算导致小数位被截掉问题，可以将其中一个操作数强制类型转换为 float，见代码第③行，这样计算过程中操作数是 float 类型，结果是 float 不会截掉小数部分。

再看一个强制类型转换与精度丢失的示例：

```
long yourNumber = 6666666666L;
System.out.println(yourNumber);
int myNuber = (int)yourNumber;
System.out.println(myNuber);
```

输出结果为：

```
6666666666
-1923267926
```

从上述代码输出结果可见，经过强制类型转换后，原本的 6666666666L 变成了负数。当大宽度数值转换为小宽度数值时，大宽度数值的高位被截掉，这样就会导致数据精度丢失。除非大宽度数值的高位没有数据，就是这个数比较小的情况，例如将 6666666666L 换为 6L 就不会丢失精度。

5.8 引用数据类型

在 Java 中除了 8 种基本数据类型外，其他数据类型全部都是引用（reference）数据类型。引用数据类型用来表示复杂数据类型，如图 5-3 所示，包含类、接口和数组声明的数据类型。

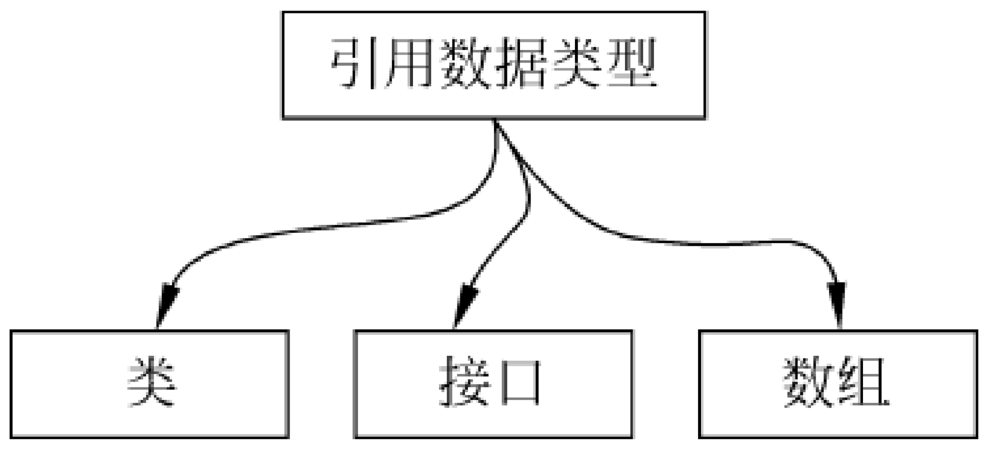


图 5-3 引用数据类型

提示 Java 中的引用类型，相当于 C 等语言中指针（pointer）类型，引用事实上就是指针，是指向一个对象的内存地址。引用类型变量中保存的是指向对象的内存地址。很多资料上提到 Java 不支持指针，事实上是不支持指针计算，而指针类型还是保留了下来，只是在 Java 中称为引用类型。

引用数据类型示例如下：

```
int x = 7;                      ①
int y = x;                      ②

String str1 = "Hello";          ③
String str2 = str1;             ④
str2 = "World";                 ⑤
```

上述代码声明了两个基本数据类型（int）和两个引用数据类型（String）。当程序执行完第②行代码后，x 值为 7，x 赋值给 y，这时 y 的值也是 7，它们的保存方式如图 5-4 所示，x 和 y 两个变量值都是 7，但是它们之间是独立的，任何一个变化都不会影响另一个。

当程序执行完第③行代码时，字符串"Hello"对象被创建，保存到内存地址 0x12345678 中，str1 是引用类型变量，它保存的是内存地址 0x12345678，这个地址指向"Hello"对象。

当程序执行完第④行代码时，str1 变量内容（0x12345678）被赋值给 str2（引用类型变量），这样 str1 和 str2 保存了相同的内存地址，都指向"Hello"对象。如图 5-4 所示，此时 str1 和 str2 本质上是引用一个对象，通过任何一个引用都可以修改对象本身。

当程序执行完第⑤行代码时，字符串"World"对象被创建，保存到内存地址 0x23455678 中，地址保存到 str2 变量中，此时，str1 和 str2 不再指向相同内存地址，如图 5-5 所示。

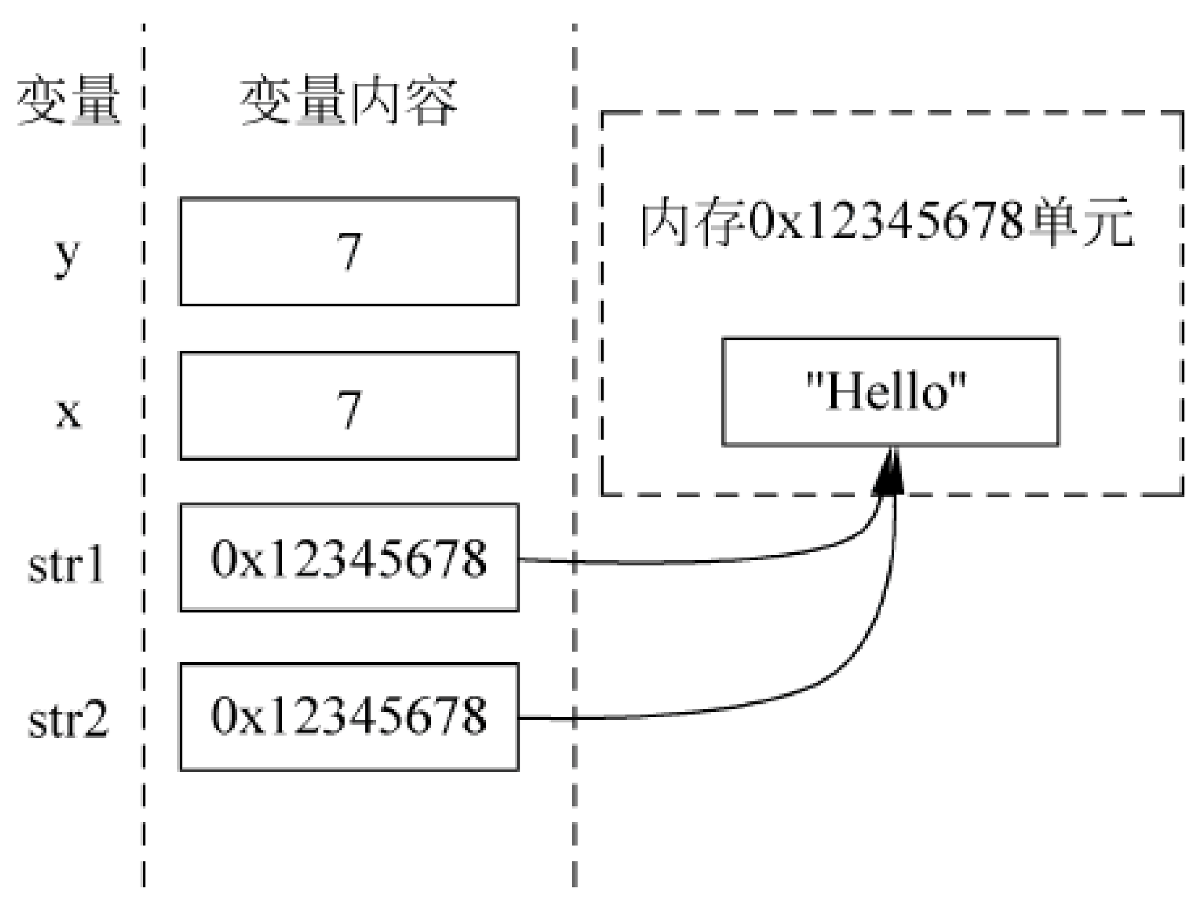


图 5-4 引用数据类型赋值过程（1）

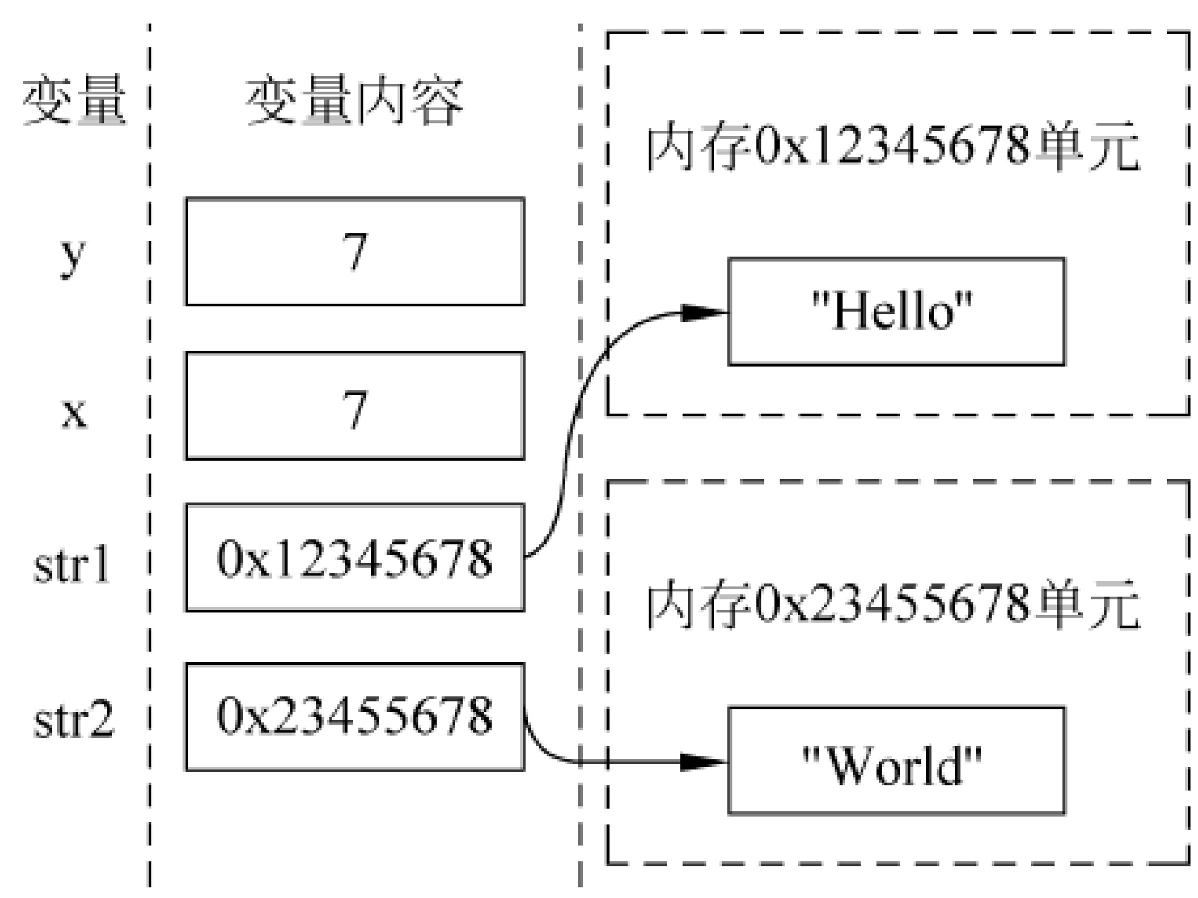


图 5-5 引用数据类型赋值过程（2）

本章小结

本章主要介绍了Java中的数据类型,读者需要重点掌握基本数据类型,理解基本数据类型与引用数据类型的区别,熟悉数值类型如何互相转换。

5.9 同步练习

1. 下面哪些行代码在编译时不会出警告或错误信息?( )

   A. float f = 1.3;　　B. char c = "a";

   C. byte b = 257;　　D. Boolean b = null;

   E. Int I = 10;

2. byte的取值范围是( )。

   A. −128 to 127　　B. −256 to 256

   C. −255 to 256　　D. 依赖于计算机本身硬件

3. 下列选项中正确的表达式有哪些?( )

   A. byte = 128;　　B. Boolean = null;

   C. long l = 0xfffL;　　D. double = 0.9239d;

4. 下列选项中( )不是Java的基本数据类型。

   A. short　　B. Boolean

   C. Int　　D. float

第6章 运算符

CHAPTER 6

Java 语言中的运算符(也称操作符)在风格和功能上都与 C 语言和 C++极为相似。本章介绍 Java 语言中一些主要的运算符,包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

6.1 算术运算符

Java 中的算术运算符主要用来组织数值类型数据的算术运算,按照参加运算的操作数的不同可以分为一元运算符和二元运算符。

6.1.1 一元运算符

算术一元运算符一共有 3 个,分别是－、＋＋和－－。具体说明如表 6-1 所示。

表 6-1 一元算术运算符

运算符	名称	说明	例子
－	取反符号	取反运算	b ＝ －a
＋＋	自加 1	先取值再加 1,或先加 1 再取值	a＋＋或＋＋a
－－	自减 1	先取值再减 1,或先减 1 再取值	a－－或－－a

表 6-1 中,－a 是对 a 取反运算,a＋＋或 a－－是在表达式运算完成后再给 a 加 1 或减 1。而＋＋a 或－－a 是先给 a 加 1 或减 1,再进行表达式运算。

示例代码如下:

```
int a = 12;
System.out.println(-a);                          ①
int b = a++;                                     ②
System.out.println(b);
b = ++a;                                         ③
System.out.println(b);
```

输出结果如下:

```
-12
12
14
```

上述代码第①行是－a，是把 a 变量取反，输出结果是－12。第②行代码是先把 a 赋值给 b 变量再加 1，即先赋值后＋＋，因此输出结果是 12。第③行代码是把 a 加 1，然后把 a 赋值给 b 变量，即先＋＋后赋值，因此输出结果是 14。

6.1.2　二元运算符

二元运算符包括＋、－、＊、/和％，这些运算符对数值类型数据都有效。具体说明如表 6-2 所示。

表 6-2　二元算术运算符

运算符	名　称	说　明	例　子
＋	加	求 a 加 b 的和，还可用于 String 类型，进行字符串连接操作	a ＋ b
－	减	求 a 减 b 的差	a － b
＊	乘	求 a 乘以 b 的积	a ＊ b
/	除	求 a 除以 b 的商	a / b
％	取余	求 a 除以 b 的余数	a ％ b

示例代码如下：

```
//声明一个字符类型变量
char charNum = 'A';
//声明一个整数类型变量
int intResult = charNum + 1;                    ①
System.out.println(intResult);

intResult = intResult - 1;
System.out.println(intResult);

intResult = intResult * 2;
System.out.println(intResult);

intResult = intResult / 2;
System.out.println(intResult);

intResult = intResult + 8;
intResult = intResult % 7;
System.out.println(intResult);

System.out.println("-------");

//声明一个浮点型变量
double doubleResult = 10.0;
System.out.println(doubleResult);

doubleResult = doubleResult - 1;
System.out.println(doubleResult);

doubleResult = doubleResult * 2;
```

```
System.out.println(doubleResult);

doubleResult = doubleResult / 2;
System.out.println(doubleResult);

doubleResult = doubleResult + 8;
doubleResult = doubleResult % 7;
System.out.println(doubleResult);
```

输出结果如下：

```
66
65
130
65
3
-------
10.0
9.0
18.0
9.0
3.0
```

上述例子中分别对数值类型数据进行了二元运算，其中代码第①行将字符类型变量charNum与整数类型进行加法运算，参与运算的该字符('A')的Unicode编码为65。其他代码比较简单，这里不再赘述。

6.1.3 算术赋值运算符

算术赋值运算符只是一种简写，一般用于变量自身的变化。具体说明如表6-3所示。

表6-3 算术赋值运算符

运算符	名称	例子
+=	加赋值	a += b、a += b+3
-=	减赋值	a -= b
*=	乘赋值	a *= b
/=	除赋值	a /= b
%=	取余赋值	a %= b

示例代码如下：

```
int a = 1;
int b = 2;
a += b;                    //相当于 a = a + b
System.out.println(a);

a += b + 3;                //相当于 a = a + b + 3
System.out.println(a);
```

```
a -= b;                    //相当于 a = a - b
System.out.println(a);

a *= b;                    //相当于 a=a*b
System.out.println(a);

a /= b;                    //相当于 a=a/b
System.out.println(a);

a %= b;                    //相当于 a=a%b
System.out.println(a);
```

输出结果如下：

```
3
8
6
12
6
0
```

上述例子分别对整型进行了+=、-=、*=、/=和%=运算，具体语句不再赘述。

6.2　关系运算符

关系运算是比较两个表达式大小关系的运算，它的结果是布尔类型数据，即 true 或 false。关系运算符有 6 种：==、!=、>、<、>=和<=，具体说明如表 6-4 所示。

表 6-4　关系运算符

运算符	名　称	说　明	例　子
==	等于	a 等于 b 时返回 true，否则返回 false。可以应用于基本数据类型和引用数据类型	a == b
!=	不等于	与==相反	a != b
>	大于	a 大于 b 时返回 true，否则返回 false，只应用于基本数据类型	a > b
<	小于	a 小于 b 时返回 true，否则返回 false，只应用于基本数据类型	a < b
>=	大于或等于	a 大于或等于 b 时返回 true，否则返回 false，只应用于基本数据类型	a >= b
<=	小于或等于	a 小于或等于 b 时返回 true，否则返回 false，只应用于基本数据类型	a <= b

提示　== 和 != 可以应用于基本数据类型和引用数据类型。当用于引用数据类型比较时，比较的是两个引用是否指向同一个对象，但在实际开发过程中多数情况下，只是比较对象的内容是否相当，不需要比较是否为同一个对象。

示例代码如下：

```
int value1 = 1;
int value2 = 2;

if (value1 == value2) {
    System.out.println("value1 == value2");
}

if (value1 != value2) {
    System.out.println("value1 != value2");
}

if (value1 > value2) {
    System.out.println("value1 > value2");
}

if (value1 < value2) {
    System.out.println("value1 < value2");
}

if (value1 <= value2) {
    System.out.println("value1 <= value2");
}
```

输出结果如下：

```
value1 != value2
value1 < value2
value1 <= value2
```

6.3 逻辑运算符

逻辑运算符是对布尔型变量进行运算，其结果也是布尔型，具体说明如表 6-5 所示。

表 6-5 逻辑运算符

运算符	名 称	说 明	例 子
!	逻辑非	a 为 true 时，值为 false；a 为 false 时，值为 true	!a
&	逻辑与	ab 全为 true 时，计算结果为 true，否则为 false	a & b
|	逻辑或	ab 全为 false 时，计算结果为 false，否则为 true	a | b
&&	短路与	ab 全为 true 时，计算结果为 true，否则为 false。&& 与 & 区别：如果 a 为 false，则不计算 b（因为不论 b 为何值，结果都为 false）	a && b
||	短路或	ab 全为 false 时，计算结果为 false，否则为 true。|| 与 | 区别：如果 a 为 true，则不计算 b（因为不论 b 为何值，结果都为 true）	a || b

提示 短路与(&&)和短路或(||)能够采用最优化的计算方式，从而提高效率。在实际编程时，应该优先考虑使用短路与和短路或。

示例代码如下：

```
int i = 0;
int a = 10;
int b = 9;

if ((a > b) || (i == 1)) {                    ①
    System.out.println("或运算为 真");
} else {
    System.out.println("或运算为 假");
}

if ((a < b) && (i == 1)) {                    ②
    System.out.println("与运算为 真");
} else {
    System.out.println("与运算为 假");
}

if ((a > b) || (a++ == --b)) {                ③
    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

上述代码运行结果如下：

```
或运算为 真
与运算为 假
a = 10, b = 9
```

其中，第①行代码进行短路计算，由于(a > b)是 true，后面的表达式(i == 1)不再计算，输出的结果为真。类似地，第②行代码也进行短路计算，由于(a < b)是 false，后面的表达式(i == 1)不再计算，输出的结果为假。

代码第③行在条件表达式中掺杂了++和--运算，由于(a > b)是 true，后面的表达式(a++ == --b)不再计算，所以最后是 a = 10，b = 9。如果把短路或(||)改为逻辑或(|)，那么输出的结果就是 a = 11，b = 8。

6.4 位运算符

位运算是以二进位(bit)为单位进行运算的，操作数和结果都是整型数据。位运算符包括 &、|、^、~、>>、<<和>>>，以及相应的赋值运算符，具体说明如表 6-6 所示。

表 6-6 位运算符

运算符	名称	例子	说明
~	位反	~x	将 x 的值按位取反
&	位与	x&y	x 与 y 位进行位与运算
|	位或	x|y	x 与 y 位进行位或运算

续表

运算符	名称	例子	说明
^	位异或	x^y	x 与 y 位进行位异或运算
>>	有符号右移	x>>a	x 右移 a 位，高位采用符号位补位
<<	左移	x<<a	x 左移 a 位，低位用 0 补位
>>>	无符号右移	x>>>a	x 右移 a 位，高位用 0 补位
&=	位与等于	a &= b	等价于 a = a&b
|=	位或等于	a |= b	等价于 a = a|b
^=	位异或等于	a ^= b	等价于 a = a^b
<<=	左移等于	a <<= b	等价于 a = a<<b
>>=	右移等于	a >>= b	等价于 a = a>>b
>>>=	右移等于	a >>>= b	等价于 a = a>>>b

注意 无符号右移>>>运算符仅被允许用在 int 和 long 整数类型，如果用于 short 或 byte 数据，则数据在位移之前转换为 int 类型后再进行位移计算。

位运算示例代码如下：

```
byte a = 0B00110010;                                    //十进制 50          ①
byte b = 0B01011110;                                    //十进制 94          ②

System.out.println("a | b = " + (a | b));               //0B01111110         ③
System.out.println("a & b = " + (a & b));               //0B00010010         ④
System.out.println("a ^ b = " + (a ^ b));               //0B01101100         ⑤
System.out.println("~b = " + (~b));                     //0B10100001         ⑥

System.out.println("a >> 2 = " + (a >> 2));             //0B00001100         ⑦
System.out.println("a >> 1 = " + (a >> 1));             //0B00011001         ⑧
System.out.println("a >>> 2 = " + (a >>> 2));           //0B00001100         ⑨
System.out.println("a << 2 = " + (a << 2));             //0B11001000         ⑩
System.out.println("a << 1 = " + (a << 1));             //0B01100100         ⑪

int c = -12;                                                                 ⑫
System.out.println("c >>> 2 = " + (c >>> 2));                                ⑬
System.out.println("c >> 2 = " + (c >> 2));                                  ⑭
```

输出结果如下：

```
a | b = 126
a & b = 18
a ^ b = 108
~b = -95
a >> 2 = 12
a >> 1 = 25
a >>> 2 = 12
a << 2 = 200
a << 1 = 100
c >>> 2 = 1073741821
c >> 2 = -3
```

上述代码第①行和第②行分别定义了 byte 变量 a 和 b，为了便于查看代码采用二进制整数表示。

代码第③行中表达式(a | b)进行位或运算，结果是二进制的 0B01111110。a 和 b 按位进行或计算，只要有一个为 1，这一位就为 1，否则为 0。

代码第④行(a & b)是进行位与运算，结果是二进制的 0B00010010。a 和 b 按位进行与计算，只有两位全部为 1，这一位才为 1，否则为 0。

代码第⑤行(a ^ b)是进行位异或运算，结果是二进制的 0B01101100。a 和 b 按位进行异或计算，只有两位相反时这一位才为 1，否则为 0。

代码第⑦行(a >> 2)是进行有符号右位移 2 位运算，结果是二进制的 0B00001100。a 的低位被移除掉，由于正数符号位是 0，高位空位用 0 补。类似代码第⑧行(a >> 1)是进行右位移 1 位运算，结果是二进制的 0B00011001。

代码第⑨行(a >>> 2)是进行无符号右位移 2 位运算，与代码第⑦行不同的是，无论是否有符号位，高位空位用 0 补，所以在正数情况下>>和>>>运算结果是一样的。

代码第⑩行(a << 2)是进行左位移 2 位运算，结果是二进制的 0B11001000。a 的高位被移除掉，低位用 0 补位。类似代码第⑪行(a << 1)是进行左位移 1 位运算，结果是二进制的 0B01100100。

代码第⑫行声明 int 类型负数。右位移(>>>和>>)在负数情况下差别比较大。代码第⑬行的(c >>> 2)表达式输出结果是 1073741821，这是一个如此大的正数，从一个负数变成一个正数，这说明无符号右位移对于负数计算会导致精度的丢失。而有符号右位移对于负数的计算是正确的，见代码第⑭行。

提示　有符号右移 n 位，相当于操作数除以 2^n，例如代码第⑦行(a >> 2)表达式相当于($a/2^2$)，a = 50，所以结果等于 12，类似的还有代码第⑧行和第⑭行。另外，左位移 n 位，相当于操作数乘以 2^n，例如代码第⑩行(a << 2)表达式相当于($a * 2^2$)，a = 50，所以结果等于 200，类似的还有代码第⑪行。

6.5　其他运算符

除了前面介绍的主要运算符，Java 还有一些其他运算符。

- 三元运算符(? :)。例如 x? y：z;，其中 x、y 和 z 都为表达式。
- 小括号。起到改变表达式运算顺序的作用，它的优先级最高。
- 中括号。数组下标。
- 引用号(.)。对象调用实例变量或实例方法的操作符，也是类调用静态变量或静态方法的操作符。
- 赋值号(=)。赋值是用等号运算符(=)进行的。
- instanceof。判断某个对象是否属于某个类。
- new。对象内存分配运算符。
- 箭头(->)。Java 8 新增加的，用来声明 Lambda 表达式。
- 双冒号(::)。Java 8 新增加的，用于 Lambda 表达式中方法的引用。

示例代码如下：

```
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) {

        int score = 80;
        String result = score > 60 ? "及格" : "不及格";     //三元运算符(? : )
        System.out.println(result);

        Date date = new Date();                            //new 运算符可以创建 Date 对象
        System.out.println(date.toString());               //通过.运算符调用方法

    }
}
```

此外,还有一些鲜为人知的运算符,随着学习的深入用到时再介绍,这里就不再赘述了。

6.6 运算符优先级

在一个表达式计算过程中,运算符的优先级非常重要。表 6-7 中从上到下,运算符的优先级从高到低,同一行具有相同的优先级。二元运算符计算顺序从左向右,但是优先级 15 的赋值运算符的计算顺序是从右向左的。

表 6-7 Java 运算符优先级

优 先 级	运 算 符
1	.(引用号)、小括号、中括号
2	++、--、-(数值取反)、~(位反)、!(逻辑非)、类型转换小括号
3	*、/、%
4	+、-
5	<<、>>、>>>
6	<、>、<=、>=、instanceof
7	==、!=
8	&(逻辑与、位与)
9	^(位异或)
10	|(逻辑或、位或)
11	&&
12	||
13	?:
14	→
15	=、*=、/=、%=、+=、-=、<<=、>>=、>>>=、&=、^=、|=

总结:运算符优先级从高到低是算术运算符→位运算符→关系运算符→逻辑运算符→赋值运算符。

本章小结

通过对本章内容的学习，读者可以了解到Java语言的基本运算符，这些运算符包括算术运算符、关系运算符、逻辑运算符、位运算符和其他运算符。

6.7　同步练习

1. 下列选项中合法的赋值语句有哪些？(　　)
   A. a == 1;　　　　B. ++ i;
   C. a = a + 1= 5;　　　　D. y = int(i);
2. 如果所有变量都已正确定义，以下选项中非法的表达式有哪些？(　　)
   A. a != 4 || b == 1　　　　B. 'a' % 3
   C. 'a' = 1/2　　　　D. 'A' + 32
3. 如果定义 int a = 2；则执行完语句 a += a -= a * a；后 a 的值是(　　)。
   A. 0　　B. 4　　C. 8　　D. -4
4. 下面关于使用"<<"和 ">>"操作符的哪些结果是正确的？(　　)
   A. 1010 0000 0000 0000 0000 0000 0000 0000 >> 4 的结果是
   0000 1010 0000 0000 0000 0000 0000 0000
   B. 1010 0000 0000 0000 0000 0000 0000 0000 >> 4 的结果是
   1111 1010 0000 0000 0000 0000 0000 0000
   C. 1010 0000 0000 0000 0000 0000 0000 0000 >>> 4 的结果是
   0000 1010 0000 0000 0000 0000 0000 0000
   D. 1010 0000 0000 0000 0000 0000 0000 0000 >>> 4 的结果是
   1111 1010 0000 0000 0000 0000 0000 0000

第 7 章 控制语句

CHAPTER 7

程序设计中的控制语句有 3 种，即分支语句、循环语句和跳转语句。Java 程序通过控制语句来管理程序流，完成一定的任务。程序流是由若干个语句组成的，语句可以是一条单一的语句，也可以是一个用大括号（{}）括起来的复合语句。Java 中的控制语句有以下几类。

- 分支语句：if 和 switch。
- 循环语句：while、do-while 和 for。
- 跳转语句：break、continue、return 和 throw。

7.1 分支语句

分支语句提供了一种控制机制，使得程序具有了“判断能力”，能够像人类的大脑一样分析问题。分支语句又称条件语句，条件语句使部分程序可根据某些表达式的值被有选择地执行。Java 编程语言提供了 if 和 switch 两种分支语句。

7.1.1 if 语句

由 if 语句引导的选择结构有 if 结构、if-else 结构和 else-if 结构。

1. if 结构

如果条件表达式为 true 就执行语句组，否则就执行 if 结构后面的语句。如果语句组只有一条语句，可以省略大括号，但从编程规范角度来看不要省略大括号，省略大括号会使程序的可读性变差。语法结构如下：

```
if (条件表达式) {
    语句组
}
```

if 结构的示例代码如下：

```
int score = 95;
if (score >= 85) {
    System.out.println("您真优秀!");
}
if (score < 60) {
```

```
    System.out.println("您需要加倍努力!");
}
if ((score >= 60) && (score < 85)) {
    System.out.println("您的成绩还可以,仍需继续努力!");
}
```

程序运行结果如下：

```
您真优秀!
```

2. if-else 结构

所有的语言都有 if-else 结构，而且结构的格式基本相同，语法结构如下：

```
if (条件表达式) {
  语句组 1
} else {
  语句组 2
}
```

当程序执行到 if 语句时，先判断条件表达式，如果值为 true，则执行语句组 1，然后跳过 else 语句及语句组 2，继续执行后面的语句；如果条件表达式的值为 false，则忽略语句组 1 而直接执行语句组 2，然后继续执行后面的语句。

if-else 结构的示例代码如下：

```
int score = 95;
if (score < 60) {
    System.out.println("不及格");
} else {
    System.out.println("及格");
}
```

程序运行结果如下：

```
及格
```

3. else-if 结构

else-if 结构的语法结构如下：

```
if (条件表达式 1) {
    语句组 1
} else if (条件表达式 2) {
    语句组 2
} else if (条件表达式 3) {
    语句组 3
…
} else if (条件表达式 n) {
    语句组 n
} else {
    语句组 n+1
}
```

可以看出，else-if 结构实际上是 if-else 结构的多层嵌套，它明显的特点就是在多个分支中只执行一个语句组，而其他分支都不执行，所以这种结构可以用于有多种判断结果的分支中。

else-if 结构的示例代码如下：

```
int testScore = 76;
char grade;
if (testScore >= 90) {
    grade = 'A';
} else if (testScore >= 80) {
    grade = 'B';
} else if (testScore >= 70) {
    grade = 'C';
} else if (testScore >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
System.out.println("Grade = " + grade);
```

程序运行结果如下：

```
Grade = C
```

其中，char grade 是声明字符变量，经过判断最后结果是 C。

7.1.2 switch 语句

switch 提供多分支程序结构语句。下面先介绍一下 switch 语句基本形式的语法结构，如下所示：

```
switch (表达式) {
    case 值 1:
        语句组 1
    case 值 2:
        语句组 2
    case 值 3:
        语句组 3
        ...
    case 判断值 n:
        语句组 n
    default:
        语句组 n+1
}
```

switch 语句中“表达式”计算结果只能是 int、byte、short、char 和 String 类型，以及枚举类型，不能是 long 等其他类型。每个 case 后面只能跟一个 int、byte、short、char、String 和枚举类型的常量，default 语句可以省略。

当程序执行到 switch 语句时，先计算条件表达式的值，假设值为 A，然后拿 A 与第 1 个

case 语句中的值 1 进行匹配，如果匹配则执行语句组 1，语句组执行完成后不跳出 switch，只有遇到 break 才跳出 switch。如果 A 没有与第 1 个 case 语句匹配，则与第 2 个 case 语句进行匹配，如果匹配则执行语句组 2，以此类推，直到执行语句组 n。如果所有的 case 语句都没有执行，就执行 default 的语句组 n+1，这时才跳出 switch。

示例代码如下：

```
int testScore = 75;

char grade;
switch(testScore / 10) {                        ①
case 9:
    grade = '优';
    break;
case 8:
    grade = '良';
    break;
case 7:              // 7 是贯通的              ②
case 6:
    grade = '中';
    break;
default:
    grade = '差';
}
System.out.println("Grade = " + grade);
```

输出结果如下：

```
Grade = 中
```

上述代码将 100 分制转换为“优”“良”“中”“差”评分制，其中 7 分和 6 分都是“中”成绩，把 case 7 和 case 6 当成一种情况考虑。代码第①行计算表达式获得 0～9 分数值。代码第②行的 case 7 是贯通的，只有它的后面不加 break，程序流执行完当前 case 后，则会进入下一个 case，因此本例中 case 7 和 case 6 都执行相同的代码。

7.2　循环语句

循环语句能够使程序代码重复执行。Java 支持 3 种循环构造类型：while、do-while 和 for。for 和 while 循环是在执行循环体之前测试循环条件，而 do-while 是在执行循环体之后测试循环条件。这就意味着 for 和 while 循环可能连一次循环体都未执行，而 do-while 将至少执行一次循环体。另外，Java 5 之后推出增强 for 循环语句，增强 for 循环是 for 循环的变形，它是专门为集合遍历而设计的。

7.2.1　while 语句

while 语句是一种先判断的循环结构，格式如下：

```
while (循环条件) {
    语句组
}
```

while 循环没有初始化语句，循环次数是不可知的，只要循环条件满足，循环就会一直进行下去。

下面看一个简单的示例。代码如下：

```
int i = 0;
while (i * i < 100000) {
    i++;
}

System.out.println("i = " + i);
System.out.println("i * i = " + (i * i));
```

输出结果如下：

```
i = 317
i * i = 100489
```

上述程序代码的目的是找到平方数小于 100 000 的最大整数。使用 while 循环需要注意几点，while 循环条件语句中只能写一个表达式，而且是一个布尔型表达式，那么如果循环体中需要循环变量，就必须在 while 语句之前对循环变量进行初始化。本例中先给 i 赋值 0，然后在循环体内部必须通过语句更改循环变量的值，否则将会发生死循环。

7.2.2 do-while 语句

do-while 语句的使用与 while 语句相似，不过 do-while 语句是事后判断循环条件结构。语句格式如下：

```
do {
    语句组
} while (循环条件)
```

do-while 循环没有初始化语句，循环次数是不可知的，无论循环条件是否满足，都会先执行一次循环体，然后再判断循环条件。如果条件满足，则执行循环体，不满足则停止循环。

下面看一个示例代码：

```
int i = 0;
do {
    i++;
} while (i * i < 100000);

System.out.println("i = " + i);
System.out.println("i * i = " + (i * i));
```

输出结果如下：

```
i = 317
i * i = 100489
```

该示例与 7.2.1 节的示例是一样的，都是找到平方数小于 100 000 的最大整数。输出结果也是一样的。

7.2.3　for 语句

for 语句是应用最广泛、功能最强的一种循环语句。一般格式如下：

```
for (初始化; 循环条件; 迭代) {
    语句组
}
```

for 语句执行流程如图 7-1 所示。首先会执行初始化语句，它的作用是初始化循环变量和其他变量，然后程序会判断循环条件是否满足，如果满足，则继续执行循环体并计算迭代语句，之后再判断循环条件，如此反复，直到判断循环条件不满足时跳出循环。

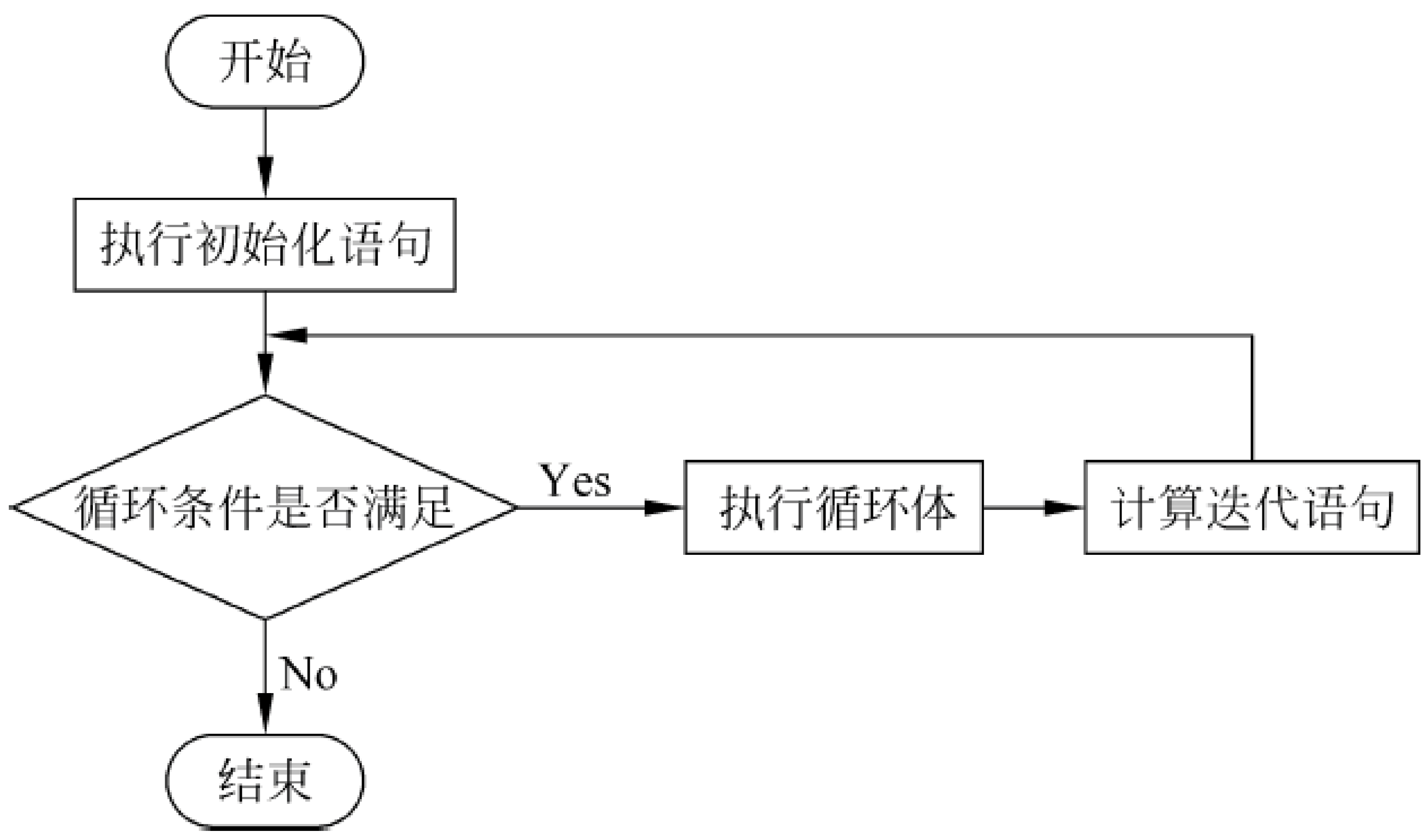


图 7-1　for 循环执行流程图

以下示例代码是计算 1～9 的平方表程序：

```
System.out.println("---------");

for (int i = 1; i < 10; i++) {
    System.out.printf("%d × %d = %d", i, i, i * i);
    //打印一个换行符，实现换行
    System.out.println();
}
```

输出结果如下：

```
---------
1 × 1 = 1
2 × 2 = 4
3 × 3 = 9
```

```
4 × 4 = 16
5 × 5 = 25
6 × 6 = 36
7 × 7 = 49
8 × 8 = 64
9 × 9 = 81
```

在这个程序的循环部分初始时，给循环变量 i 赋值为 1，每次循环都要判断 i 的值是否小于 10，如果为 true，则执行循环体，然后给 i 加 1。因此，最后的结果是打印出 1～9 的平方，不包括 10。

初始化、循环条件以及迭代部分都可以为空语句(但分号不能省略)，三者均为空的时候，相当于一个无限循环。代码如下：

```
for (; ;) {
    …
}
```

另外，在初始化部分和迭代部分，可以使用逗号语句来进行多个操作。逗号语句是用逗号分隔的语句序列，如下面程序代码所示：

```
int x;
int y;

for (x = 0, y = 10; x < y; x++, y--) {
    System.out.printf("(x,y) = (%d, %d)", x, y);
    //打印一个换行符,实现换行
    System.out.println();
}
```

输出结果如下：

```
(x,y) = (0,10)
(x,y) = (1,9)
(x,y) = (2,8)
(x,y) = (3,7)
(x,y) = (4,6)
```

7.2.4 增强 for 循环语句

Java 5 之后提供了一种专门用于遍历集合的 for 循环——增强 for 循环。使用增强 for 循环不必按照 for 的标准套路编写代码，只需要提供一个集合就可以遍历。

假设有一个数组，采用 for 语句遍历数组的方式如下：

```
//声明并初始化 int 数组
int[] numbers = { 43, 32, 53, 54, 75, 7, 10 };

System.out.println("----for-------");
```

```
//for 语句
for (int i = 0; i < numbers.length; i++) {
    System.out.println("Count is:" + numbers[i]);
}
```

上述语句 int[] numbers = { 43，32，53，54，75，7，10 }声明并初始化了 7 个元素数组，目前只需要知道当初始化数组时，要把相同类型的元素放到{…}中并且用逗号(，)分隔即可。关于数组会在后面第 8 章详细介绍。numbers.length 是获得数组的长度，length 是数组的属性，numbers[i]是通过数组下标访问数组元素的。

采用增强 for 循环语句遍历数组的方式如下：

```
//声明并初始化 int 数组
int[] numbers = { 43, 32, 53, 54, 75, 7, 10 };

System.out.println("---- 增强 for ----");
//增强 for 语句
for (int item : numbers) {
    System.out.println("Count is:" + item);
}
```

从示例中可以发现，item 不是循环变量，它保存了集合中的元素，增强 for 语句将集合中的元素一一取出来，并保存到 item 中，这个过程中不需要使用循环变量，通过数组下标访问数组中的元素。可见增强 for 语句在遍历集合的时候要简单方便得多。

7.3 跳转语句

跳转语句能够改变程序的执行顺序，可以实现程序的跳转。Java 有 4 种跳转语句：break、continue、throw 和 return。本节重点介绍 break 和 continue 语句的使用。throw 和 return 将在后面章节介绍。

7.3.1 break 语句

break 语句可用于上一节介绍的 while、do-while 和 for 循环结构，它的作用是强行退出循环体，不再执行循环体中剩余的语句。

在循环体中使用 break 语句有两种方式：带有标签和不带标签。语法格式如下：

```
break;                    //不带标签
break label;              //带标签,label 是标签名
```

不带标签的 break 语句使程序跳出所在层的循环体，而带标签的 break 语句使程序跳出标签指示的循环体。

下面看一个示例，代码如下：

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for(int i = 0; i < numbers.length; i++) {
```

```
    if(i == 3) {
        //跳出循环
        break;
    }
    System.out.println("Count is: " + i);
}
```

在上述程序代码中，当条件 i==3 的时候执行 break 语句，break 语句会终止循环。程序运行的结果如下：

```
Count is: 0
Count is: 1
Count is: 2
```

break 语句还可以配合标签使用，示例代码如下：

```
label1: for (int x = 0; x < 5; x++) {                          ①
    for (int y = 5; y > 0; y--) {                              ②
        if(y == x) {
            //跳转到 label1 指向的循环
            break label1;                                      ③
        }
        System.out.printf("(x,y) = (%d, %d)", x, y);
        //打印一个换行符，实现换行
        System.out.println();
    }
}
System.out.println("Game Over!");
```

默认情况下，break 只会跳出最近的内循环（代码第②行的 for 循环）。如果要跳出代码第①行的外循环，可以为外循环添加一个标签 label1，注意，在定义标签的时候后面跟一个冒号（:）。代码第③行的 break 语句后面指定了 label1 标签，这样，当条件满足执行 break 语句时，程序就会跳转出 label1 标签所指定的循环。

程序运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
Game Over!
```

如果 break 后面没有指定外循环标签，则运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
(x,y) = (2,5)
(x,y) = (2,4)
(x,y) = (2,3)
(x,y) = (3,5)
(x,y) = (3,4)
(x,y) = (4,5)
Game Over!
```

比较两种运行结果就会发现给 break 添加标签的意义，添加标签对于多层嵌套循环是很有必要的，适当使用可以提高程序的执行效率。

7.3.2 continue 语句

continue 语句用来结束本次循环，跳过循环体中尚未执行的语句，接着进行终止条件的判断，以决定是否继续循环。对于 for 语句，在进行终止条件的判断前，还要先执行迭代语句。

在循环体中使用 continue 语句有两种方式：带有标签和不带标签。语法格式如下：

```
continue                    //不带标签
continue label              //带标签，label 是标签名
```

下面看一个示例，代码如下：

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for (int i = 0; i < numbers.length; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println("Count is: " + i);
}
```

在上述程序代码中，当条件 i==3 的时候执行 continue 语句，continue 语句会终止本次循环，循环体中 continue 之后的语句将不再执行，接着进行下次循环，所以输出结果中没有 3。

程序运行结果如下：

```
Count is: 0
Count is: 1
```

```
Count is: 2
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
```

带标签的 continue 语句示例代码如下：

```
label1: for (int x = 0; x < 5; x++) {                    ①
    for (int y = 5; y > 0; y-- ) {                       ②
        if (y == x) {
           continue label1;                              ③
        }
        System.out.printf("(x,y) = (%d, %d)", x, y);
        System.out.println();
    }
}
System.out.println("Game Over!");
```

默认情况下，continue 只会跳出最近的内循环（代码第②行的 for 循环），如果要跳出代码第①行的外循环，可以为外循环添加一个标签 label1，然后在第③行的 continue 语句后面指定这个标签 label1，这样，当条件满足执行 continue 语句时，程序就会跳转出外循环。

程序运行结果如下：

```
(x,y) = (0,5)
(x,y) = (0,4)
(x,y) = (0,3)
(x,y) = (0,2)
(x,y) = (0,1)
(x,y) = (1,5)
(x,y) = (1,4)
(x,y) = (1,3)
(x,y) = (1,2)
(x,y) = (2,5)
(x,y) = (2,4)
(x,y) = (2,3)
(x,y) = (3,5)
(x,y) = (3,4)
(x,y) = (4,5)
Game Over!
```

由于跳过了 x == y，因此下面的内容没有输出：

```
(x,y) = (1,1)
(x,y) = (2,2)
(x,y) = (3,3)
(x,y) = (4,4)
```

本章小结

通过对本章内容的学习，可以了解到 Java 语言的控制语句，其中包括分支语句（if 和 switch）、循环语句（while、do-while、for 和增强 for）和跳转语句（break 和 continue）等。

7.4 同步练习

1. 编程题：水仙花数是一个三位数，三位数各位的立方之和等于三位数本身。

（1）请使用 for 循环计算水仙花数。

（2）请使用 while 循环计算水仙花数。

2. 编程题：编写程序以输出以下形式的金字塔图案。

```
   *
  ***
 *****
*******
```

3. 能从循环语句的循环体中跳出的语句是（　　）。

A. for 语句　　B. break 语句　　C. while 语句　　D. continue 语句

4. 下列语句执行后，x 的值是（　　）。

```
int a = 3, b = 4, x = 5;

if (a < b) {
    a++;
    ++x;
}
```

A. 5　　B. 3　　C. 4　　D. 6

5. 以下 Java 代码编译运行后，（　　）会出现在输出结果中。

```
public class HelloWorld {
    public static void main(String args[]) {
        for (int i = 0; i < 3; i++) {
            for (int j = 3; j >= 0; j--) {
                if (i == j)
                    continue;
                System.out.println("i = " + i + " j = " + j);
            }
        }
    }
}
```

A. i=0 j=3　　B. i=0 j=0　　C. i=2 j=2　　D. i=0 j=2

E. i=0 j=1

6. 运行下列 Java 代码后,(　　)包含在输出结果中。

```
public class HelloWorld {
    public static void main(String args[]) {
        int i = 0;
        do {
            System.out.println("i = " + i);
        } while ( -- i > 0);
        System.out.println("完成");
    }
}
```

A. i = 3　　B. i = 1　　C. i = 0　　D. 完成

第 8 章 数　组

CHAPTER 8

在计算机语言中数组是非常重要的集合类型，大部分计算机语言中数组具有如下 3 个基本特性。

(1) 一致性。数组只能保存相同数据类型元素，元素的数据类型可以是任何相同的数据类型。

(2) 有序性。数组中的元素是有序的，通过下标访问。

(3) 不可变性。数组一旦初始化，则长度（数组中元素的个数）不可变。

在 Java 语言中数组的下标是从零开始的，事实上很多计算机语言的数组下标都是从零开始的。Java 数组下标访问运算符是中括号，如 intArray[0]，表示访问 intArray 数组的第一个元素，0 是第一个元素的下标。

另外，Java 中的数组本身是引用数据类型，它的长度属性是 length。数组可以分为一维数组和多维数组。下面先介绍一维数组。

8.1 一维数组

当数组中每个元素都只带有一个下标时，这种数组就是一维数组。数组是引用数据类型，引用数据类型在使用之前一定要做两件事情：声明和初始化。

8.1.1 数组声明

数组的声明就是宣告这个数组中的元素类型，即数组的变量名。

数组声明语法格式如下：

```
元素数据类型[] 数组变量名;
元素数据类型 数组变量名[];
```

可见数组的声明有两种形式：一种是中括号（[]）跟在元素数据类型之后；另一种是中括号（[]）跟在变量名之后。

注意 数组声明完成后，数组的长度还不能确定，JVM（Java 虚拟机）还没有给元素分配内存空间。

提示 从面向对象角度看，Java 更推荐采用第一种声明方式，因为它把“元素数据类型[]”看作是一个整体类型，即数组类型。而第二种是 C 语言数组声明方式。

数组声明示例如下：

```
int intArray[];
float[] floatArray;
String strArray[];
Date[] dateArray;
```

8.1.2 数组初始化

声明完成就要对数组进行初始化，数组初始化的过程就是为数组每一个元素分配内存空间，并为每一个元素提供初始值。初始化之后数组的长度确定下来，就不能再变化了。

提示　有些计算机语言提供了可变类型数组，即它的长度是可变的，这种数组本质上是创建了一个新的数组对象，并非是原始数组的长度发生了变化。

数组初始化可以分为静态初始化和动态初始化。

1. 静态初始化

静态初始化就是将数组的元素放到大括号中，元素之间用逗号(,)分隔。示例代码如下：

```
//静态初始化 int 数组
int[] intArray = {21,32,43,45};

//静态初始化 String 数组
string[] strArray = {"张三","李四","王五","董六"};

//声明同时初始化数组
int intArray[] = {21,32,43,45};
String strArray[] = {"张三","李四","王五","董六"};
```

静态初始化是在已知数组的每一个元素内容情况下使用的。很多情况下数据是从数据库或网络中获得的，在编程时不知道元素有多少，更不知道元素的内容，此时可采用动态初始化。

2. 动态初始化

动态初始化使用 new 运算符分配指定长度的内存空间，语法格式如下：

```
new 元素数据类型[数组长度];
```

示例代码如下：

```
int intArray[];
//动态初始化 int 数组
intArray = new int[4];                   ①
intArray[0] = 21;
intArray[1] = 32;
intArray[2] = 43;
intArray[3] = 45;                        ②
```

```
    //动态初始化 String 数组
    String[] stringArray = new String[4];        ③
    //初始化数组中元素
    stringArray[0] = "张三";
    stringArray[1] = "李四";
    stringArray[2] = "王五";
    stringArray[3] = "董六";                      ④
```

上述代码第①行和第③行通过 new 运算符分配了 4 个元素的内存空间。

当代码第①行执行完成后，intArray 数组内容如图 8-1(a)所示，intArray 数组中的所有元素都是 0，根据需要会动态添加元素内容。代码第②行执行完成，intArray 数组内容如图 8-1(b)所示。

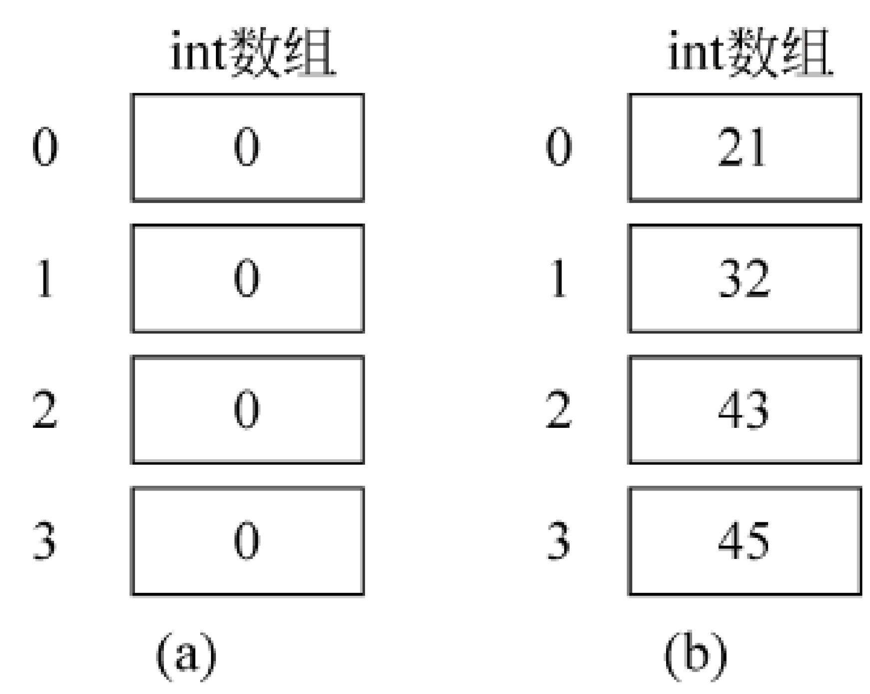


图 8-1　intArray 数组

当代码第③行执行完成后，stringArray 数组内容如图 8-2(a)所示，stringArray 数组中所有元素都是 null，随着每一个元素被初始化和赋值，代码第④行执行完之后每个元素都有不同内容，这里需要注意的是引用类型数组，每一个元素保存都是指向实际对象的内存地址，如图 8-2(b)所示，每个对象还需要有创建和初始化过程。有关对象创建和初始化内容，将在后面章节详细介绍。

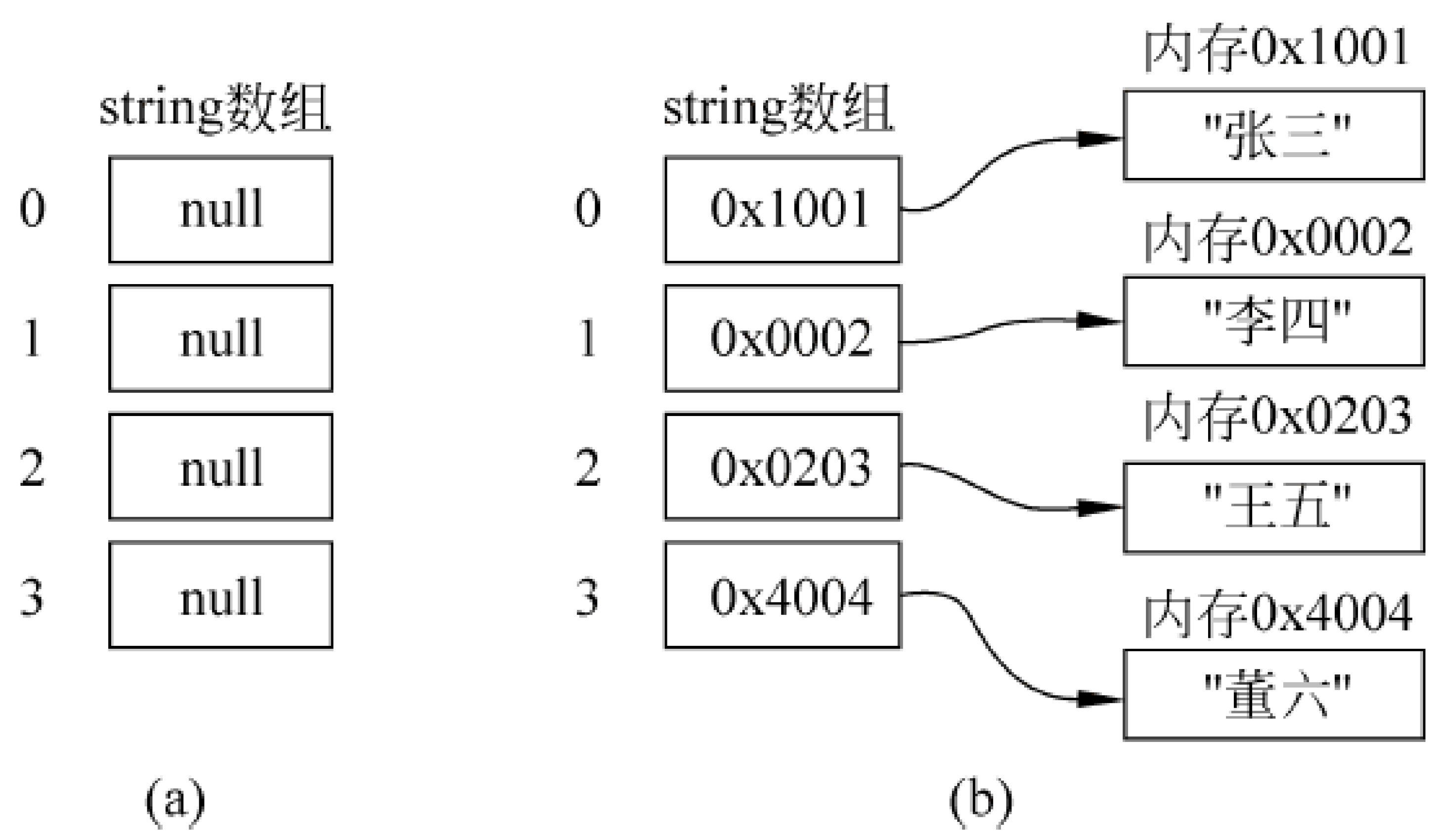


图 8-2　stringArray 数组

提示　new 分配数组内存空间后，数组中的元素内容是什么呢？答案是数组类型的默认值，不同类型默认值是不同的，如表 8-1 所示。

表 8-1　数据类型默认值

基本类型	默认值	基本类型	默认值
byte	0	double	0.0d
short	0	char	'\u0000'
int	0	boolean	false
long	0L	引用	null
float	0.0f		

8.1.3 案例：数组合并

数组长度是不可变的。要想合并两个不同的数组，不能通过在一个数组的基础上追加另一个数组实现，需要创建一个新的数组，新数组长度是两个数组长度之和。然后将两个数组的内容导入到新数组中。

具体看看如下实现代码：

```
public class HelloWorld {

    public static void main(String[] args) {

        //两个待合并数组
        int array1[] = { 20, 10, 50, 40, 30 };
        int array2[] = { 1, 2, 3 };

        //动态初始化数组，设置数组的长度是 array1 和 array2 长度之和
        int array[] = new int[array1.length + array2.length];

        //循环添加数组内容
        for(int i = 0; i < array.length; i++) {

            if(i < array1.length) {                         ①
                array[i] = array1[i];                       ②
            } else {
                array[i] = array2[i - array1.length];       ③
            }
        }

        System.out.println("合并后:");
        for(int element : array) {
            System.out.printf(" %d ", element);
        }

    }
}
```

上述代码第①行是判断当前循环变量 i 是否小于 array1.length，在此条件下将 array1 数组内容导入新数组，见代码第②行。当 array1 数组内容导入完成后，再通过代码第③行将另一个数组 array2 导入到新数组，其中 array2 下标应该是 i−array1.length。

8.2 多维数组

当数组中每个元素又可以带有多个下标时，这种数组就是多维数组。本节重点介绍二维数组。

8.2.1 二维数组声明

Java 中声明二维数组需要有两个中括号，具体有如下 3 种语法格式：

```
元素数据类型[][] 数组变量名;
元素数据类型 数组变量名[][];
元素数据类型[] 数组变量名[];
```

3 种语法中前两种比较好理解，最后一种形式看起来有些古怪。数组声明示例如下：

```
int[][] array1;
int array1[][];
int[] array1[];
```

8.2.2 二维数组的初始化

二维数组的初始化也可以分为静态初始化和动态初始化。

1. 静态初始化

静态初始化示例如下：

```
int intArray[][] = { { 1, 2, 3 }, { 11, 12, 13 }, { 21, 22, 23 }, { 31, 32, 33 } };
```

上述代码创建并初始化了一个 4×3 二维数组，理解 Java 中的多维数组应该从数组的数组的角度出发。首先将 intArray 看作是一个一维数组，它有 4 个元素，如图 8-3 所示，其中第 1 个元素是{1,2,3}，第 2 个元素是{11,12,13}，第 3 个元素是{21,22,23}，第 4 个元素是{31,32,33}。再分别考虑每一个元素，{1,2,3}表示形式说明它是一个 int 类型的一维数组，其他 3 个元素也是一维 int 类型数组。

0	{1, 2, 3}
1	{11, 12, 13}
2	{21, 22, 23}
3	{31, 32, 33}

图 8-3 intArray 二维数组

提示 严格意义上说，Java 中并不存在真正意义上的多维数组，只是一维数组，不过数组中的元素也是数组，以此类推，三维数组就是数组的数组的数组，例如{{{1,2},{3}},{{21},{22,23}}}表示一个三维数组。

2. 动态初始化

动态初始化二维数组语法格式如下：

```
new 元素数据类型[高维数组长度] [低维数组长度];
```

高维数组就是最外面的数组，低维数组是每一个元素的数组。动态创建并初始化一个 4×3 二维数组示例代码如下：

```
int[][] intArray = new int[4][3];
```

二维数组的下标有[4][3]两个，前面的[4]是高维数组下标索引，后面的[3]是低维数组下标索引。4×3 二维数组的每一个元素的下标索引如图 8-4(a)所示。由于低维数组是 int 类型，所以初始化完成后所有元素全部是 0，如图 8-4(b)所示。

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]
[3][0]	[3][1]	[3][2]

(a)

0	0	0
0	0	0
0	0	0
0	0	0

(b)

图 8-4　二维数组动态初始化

二维数组示例如下：

```
public class HelloWorld {

    public static void main(String[] args) {

        //静态初始化二维数组
        int[][] intArray = {
                { 1, 2, 3 },
                { 11, 12, 13 },
                { 21, 22, 23 },
                { 31, 32, 33 } };

        //动态初始化二维数组
        double[][] doubleArray = new double[4][3];

        //计算数组 intArray 元素的平方根,结果保存到 doubleArray
        for(int i = 0; i < intArray.length; i++) {
            for(int j = 0; j < intArray[i].length; j++) {
                //计算平方根
                doubleArray[i][j] = Math.sqrt(intArray[i][j]);       ①
            }
        }

        //打印数组 doubleArray
        for(int i = 0; i < doubleArray.length; i++) {
            for(int j = 0; j < doubleArray[i].length; j++) {
                System.out.printf("[%d][%d] = %f", i, j, doubleArray[i][j]);
                System.out.print('\t');
            }
            System.out.println();
        }
    }
}
```

代码第①行中 Math.sqrt(intArray[i][j])表达式是计算平方根，Math 是 java.lang 包中提供的用于数学计算类，它提供很多常用的数学计算方法，sqrt 是计算平方根，如取绝对值的 abs、幂运算的 pow 等。

8.2.3　不规则数组

由于 Java 多维数组是数组的数组，因此会衍生出一种不规则数组，规则的 4×3 二维数组有 12 个元素，而不规则数组就不一定了。如下代码静态初始化了一个不规则数组：

```
int intArray[][] = { { 1, 2 }, { 11 }, { 21, 22, 23 }, { 31, 32, 33 } };
```

高维数组是 4 个元素，但是低维数组元素个数不同，如图 8-5 所示，其中第 1 个数组有两个元素，第 2 个数组有 1 个元素，第 3 个数组有 3 个元素，第 4 个数组有 3 个元素，这就是不规则数组。

动态初始化不规则数组比较麻烦，不能使用 new int[4][3]语句，而是先初始化高维数组，然后再分别逐个初始化低维数组，代码如下：

```
int intArray[][] = new int[4][];          //先初始化高维数组为 4
//逐一初始化低维数组
intArray[0] = new int[2];
intArray[1] = new int[1];
intArray[2] = new int[3];
intArray[3] = new int[3];
```

上述代码初始化数组完成之后，不是 12 个元素而是 9 个元素，它们的下标索引如图 8-6 所示，可见其中下标[0][2]、[1][1]和[1][2]是不存在的，如果试图访问它们则会抛出下标越界异常。

0	{1, 2}
1	{11}
2	{21, 22, 23}
3	{31, 32, 33}

图 8-5　不规则数组

[0][0]	[0][1]	
[1][0]		
[2][0]	[2][1]	[2][2]
[3][0]	[3][1]	[3][2]

图 8-6　不规则数组访问

提示　下标越界异常（ArrayIndexOutOfBoundsException）是试图访问不存在的下标时引发的。例如，一个一维 array 数组如果有 10 个元素，那么表达式 array[10]就会发生下标越界异常，这是因为数组下标是从 0 开始的，最后一个元素下标是数组长度减 1，所以 array[10]访问的元素是不存在的。

下面介绍一个不规则数组的示例：

```
public class HelloWorld {

    public static void main(String[] args) {

        int intArray[][] = new int[4][];          //先初始化高维数组为 4
        //逐一初始化低维数组
        intArray[0] = new int[2];
        intArray[1] = new int[1];
        intArray[2] = new int[3];
```

```
            intArray[3] = new int[3];

            //for 循环遍历
            for (int i = 0; i < intArray.length; i++) {
                for (int j = 0; j < intArray[i].length; j++) {
                    intArray[i][j] = i + j;
                }
            }
            //增强 for 循环遍历
            for (int[] row : intArray) {                              ①
                for (int column : row) {                              ②
                    System.out.print(column);
                    //在元素之间添加制表符
                    System.out.print('\t');
                }
                //一行元素打印完成后换行
                System.out.println();
            }

            //System.out.println(intArray[0][2]);    //发生运行期错误      ③
        }
    }
```

不规则数组访问和遍历可以使用 for 和增强 for 循环，但要注意下标越界异常发生。上述代码第①行和第②行采用增强 for 循环遍历不规则数组，其中代码第①行增强 for 循环取出的数据是 int 数组，所以 row 的类型是 int[]。代码第②行增强 for 循环取出的数据是 int 数据，所以 column 的类型是 int。

另外，注意代码第③行试图访问 intArray[0][2]元素，由于[0][2]不存在，所以会发生下标越界异常。

本章小结

本章介绍了 Java 的数组，包括一维数组和多维数组，读者要重点掌握一维数组的声明、初始化和使用，了解二维数组的声明、初始化和使用。另外，还需要了解不规则数组。

8.3 同步练习

1. 执行代码 String[] s=new String[10]；后，下列选项中(　　)正确的。
   A. s[10]为""；　　　　B. s[9]为 null；
   C. s[0]为未定义　　　　D. s.length 为 10
2. 以下定义一维数组的语句中正确的是(　　)。
   A. int a[5]　　　　B. int a[]=new [5]；
   C. int a[]；int a=new int[5]；　　　　D. int a[]={1,2,3,4,5}；

3. 假设有定义语句 int a[]={66,88,99}；则以下对此语句的叙述错误的是(　　)。

A. 定义了一个名为 a 的一维数组　　B. a 数组有 3 个元素

C. a 数组的下标为 1～3　　D. 数组中的每个元素是整型

4. 为了定义 3 个整型数组 a1、a2、a3，下面声明正确的语句是(　　)。

A. int Array [] a1,a2; int a3[]={1,2,3,4,5};

B. int[] a1,a2; int a3[]={1,2,3,4,5};

C. int a1,a2[]; int a3={1,2,3,4,5};

D. int [] a1,a2; int a3=(1,2,3,4,5);

5. 在一个应用程序中有如下定义 int a[]={1,2,3,4,5,6,7,8,9,10}；，为了打印输出数组 a 的最后一个元素，下面正确的代码是(　　)。

A. System.out.println(a[10]);　　B. System.out.println(a[9]);

C. System.out.println(a[a.length-1]);　　D. System.out.println(a(8));

第9章 字 符 串

CHAPTER 9

由字符组成的一串字符序列称为字符串,在前面的章节中也多次用到了字符串,本章将对其重点介绍。

9.1 Java 中的字符串

Java 中的字符串是由双引号括起来的多个字符,下面示例都是表示字符串常量:

```
"Hello World"                                                            ①
"\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\u0064"     ②
"世界你好"                                                               ③
"A"                                                                      ④
""                                                                       ⑤
```

Java 中的字符采用 Unicode 编码,所以 Java 字符串可以包含中文等亚洲字符,见代码第③行的"世界你好"字符串。代码第②行的字符串是用 Unicode 编码表示的字符串,事实上它表示的也是"Hello World"字符串,可通过 System.out.print 方法将 Unicode 编码表示的字符串输出到控制台,则会看到 Hello World 字符串。

另外,单个字符如果用双引号括起来,则其表示的是字符串,而不是字符了,见代码第④行的"A"是表示字符串 A,而不是字符 A。

Java SE 提供了 3 个字符串类:String、StringBuffer 和 StringBuilder。String 是不可变字符串,StringBuffer 和 StringBuilder 是可变字符串。

注意 字符串还有一个极端情况,就是代码第⑤行的" "表示空字符串,双引号中没有任何内容,空字符串不是 null,空字符串是分配内存空间,而 null 是没有分配内存空间。

9.2 使用 API 文档

Java 中有很多类,每一个类又有很多方法和变量,通过查看 Java API 文档能够知道这些类、方法和变量如何使用。作为 Java 程序员应该熟悉如何使用 API 文档。

本节介绍一下如何使用 Java SE 的 API 文档。Java 官方提供了 Java 8 在线 API 文档，网址是 http://docs.oracle.com/javase/8/docs/api/，页面如图 9-1 所示。

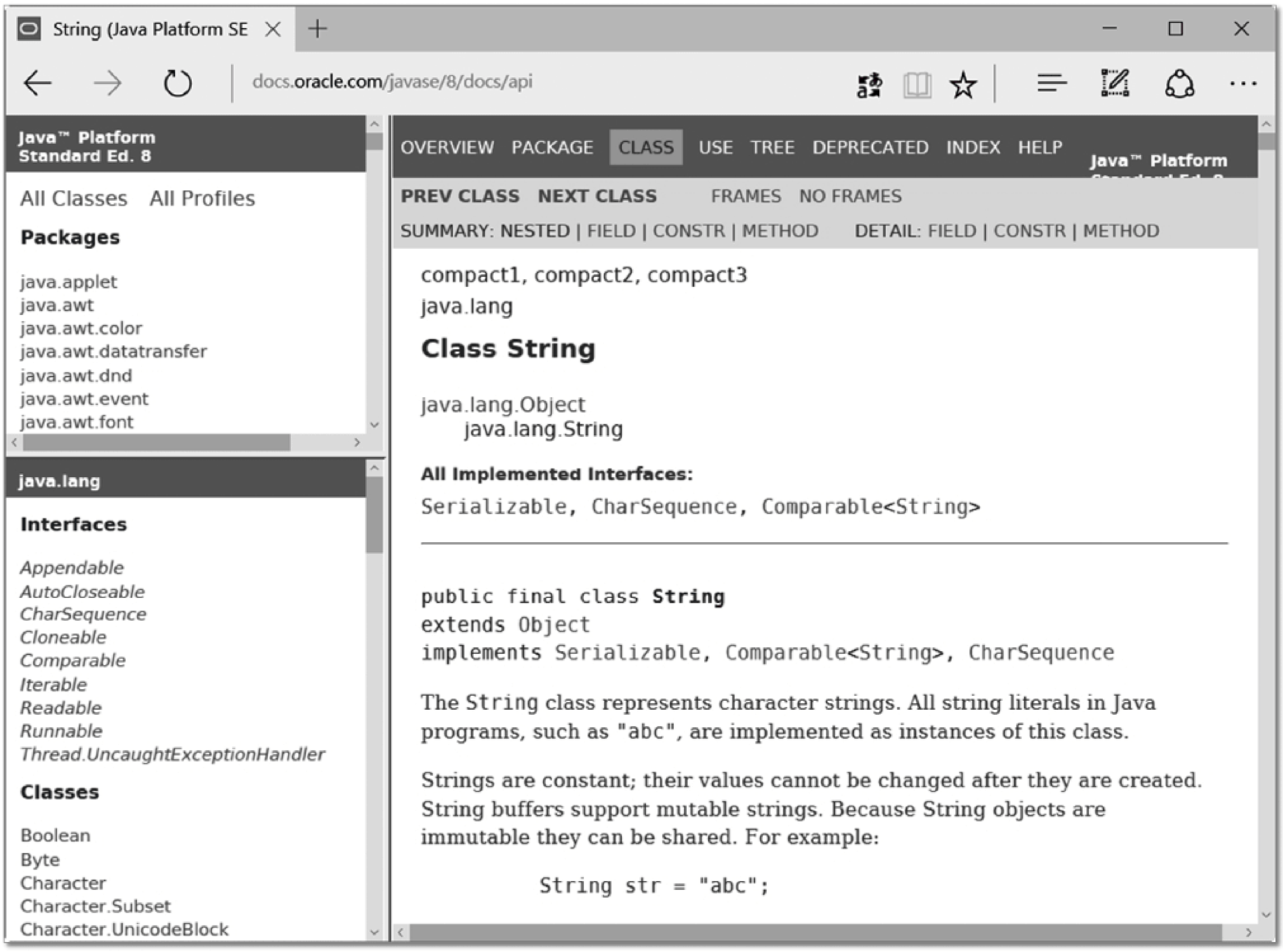


图 9-1　Java 8 在线 API 文档

下面介绍一下如何使用 API 文档，熟悉一下 API 文档页面中的各个部分含义，如图 9-2 所示，斜体显示的是接口，正常字体显示的才是类。

提示　很多读者希望能够有离线的中文 Java API 文档，但 Java 官方只提供了 Java 6 的中文 API 文档，该文件下载地址是 http://download.oracle.com/technetwork/java/javase/6/docs/zh/api.zip，下载完成后解压 api.zip 文件，找到其中的 index.html 文件，双击即可在浏览器中打开 API 文档。

在类窗口还有很多内容，向下拖曳滚动条会看到如图 9-3 所示的页面，其中“字段摘要”描述了类中的实例变量和静态变量；“构造方法摘要”描述了类中所有构造方法；“方法摘要”描述了类中所有方法。这些“摘要”只是一个概要说明，单击链接可以看到该主题更加详细的描述。图 9-4 为单击 compareTo 方法看到的详细信息。

查询 API 的一般流程为：找包→找类或接口→查看类或接口→找方法或变量。读者可以尝试查找 String、StringBuffer 和 StringBuilder 这些字符串类的 API 文档，熟悉一下这些类的用法。



图 9-2 API 文档页面各个部分



图 9-3 类窗口页面的其他内容



图 9-4 compareTo 方法详细描述

9.3 不可变字符串

很多计算机语言都提供了两种字符串，即不可变字符串和可变字符串，它们的区别在于当字符串进行拼接等修改操作时，不可变字符串会创建新的字符串对象，而可变字符串不会创建新对象。

9.3.1 String

Java 中不可变字符串类是 String，属于 java.lang 包，它也是 Java 非常重要的类。

创建 String 对象可以通过构造方法实现，常用的构造方法如下。

- String()：使用空字符串创建并初始化一个新的 String 对象。
- String(String original)：使用另外一个字符串创建并初始化一个新的 String 对象。

提示 java.lang 包中提供了很多 Java 基础类，包括 Object、Class、String 和 Math 等基本类。在使用 java.lang 包中的类时不需要引入(import)该包，因为它是由解释器自动引入的。当然，引入 java.lang 包程序也不会有编译错误。

- String(StringBuffer buffer)：使用可变字符串对象(StringBuffer)创建并初始化一个新的 String 对象。

- String(StringBuilder builder)：使用可变字符串对象(StringBuilder)创建并初始化一个新的 String 对象。
- String(byte[] bytes)：使用平台的默认字符集解码指定的 byte 数组，通过 byte 数组创建并初始化一个新的 String 对象。
- String(char[] value)：通过字符数组创建并初始化一个新的 String 对象。
- String(char[] value, int offset, int count)：通过字符数组的子数组创建并初始化一个新的 String 对象。其中 offset 参数是子数组第一个字符的索引；count 参数指定子数组的长度。

创建字符串对象示例代码如下：

```
//创建字符串对象
String s1 = new String();
String s2 = new String("Hello World");
String s3 = new String ("\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\
                    u0064");
System.out.println("s2 = " + s2);
System.out.println("s3 = " + s3);

char chars[] = { 'a', 'b', 'c', 'd', 'e' };
//通过字符数组创建字符串对象
String s4 = new String(chars);
//通过字符数组创建字符串对象
String s5 = new String(chars, 1, 4);
System.out.println("s4 = " + s4);
System.out.println("s5 = " + s5);

byte bytes[] = { 97, 98, 99 };
//通过 byte 数组创建字符串对象
String s6 = new String(bytes);
System.out.println("s6 = " + s6);
System.out.println("s6 字符串长度 = " + s6.length());
```

输出结果如下：

```
s2 = Hello World
s3 = Hello World
s4 = abcde
s5 = bcde
s6 = abc
s6 字符串长度 = 3
```

上述代码中 s2 和 s3 都是表示 Hello World 字符串，获得字符串长度方法是 length()，其他代码比较简单，这里不再赘述。

9.3.2 字符串池

在前面的学习过程中细心的读者可能会发现，前面的示例代码中获得字符串对象时都是直接使用字符串常量，但 Java 中对象是使用 new 关键字创建，字符串对象也可以使用

new 关键字创建，代码如下：

```
String s9 = "Hello";                    //字符串常量
String s7 = new String("Hello");        //使用 new 关键字创建
```

使用 new 关键字与字符串常量都能获得字符串对象，但它们之间有一些区别。先看下面代码的运行结果：

```
String s7 = new String("Hello");            ①
String s8 = new String("Hello");            ②

String s9 = "Hello";                        ③
String s10 = "Hello";                       ④

System.out.printf("s7 == s8 : %b%n", s7 == s8);
System.out.printf("s9 == s10: %b%n", s9 == s10);
System.out.printf("s7 == s9 : %b%n", s7 == s9);
System.out.printf("s8 == s9 : %b%n", s8 == s9);
```

运行结果如下：

```
s7 == s8 : false
s9 == s10: true
s7 == s9 : false
s8 == s9 : false
```

==运算符比较的是两个引用是否指向相同的对象，从上面的运行结果可见，s7 和 s8 指的是不同对象，s9 和 s10 指的是相同对象。

这是为什么？Java 中的不可变字符串 String 常量采用字符串池(String Pool)管理技术，字符串池是一种字符串驻留技术。采用字符串常量赋值时(见代码第③行)，会在字符串池中查找"Hello"字符串常量，如果已经存在，则把引用赋值给 s9，否则创建"Hello"字符串对象，并放到池中，如图 9-5 所示。根据此原理，可以推定 s10 与 s9 是相同的引用，指向同一个对象。但此原理并不适用于 new 所创建的字符串对象，代码运行到第①行后，会创建"Hello"字符串对象，而它并没有放到字符串池中。代码第②行又创建了一个新的"Hello"字符串对象，s7 和 s8 是不同的引用，指向不同的对象。

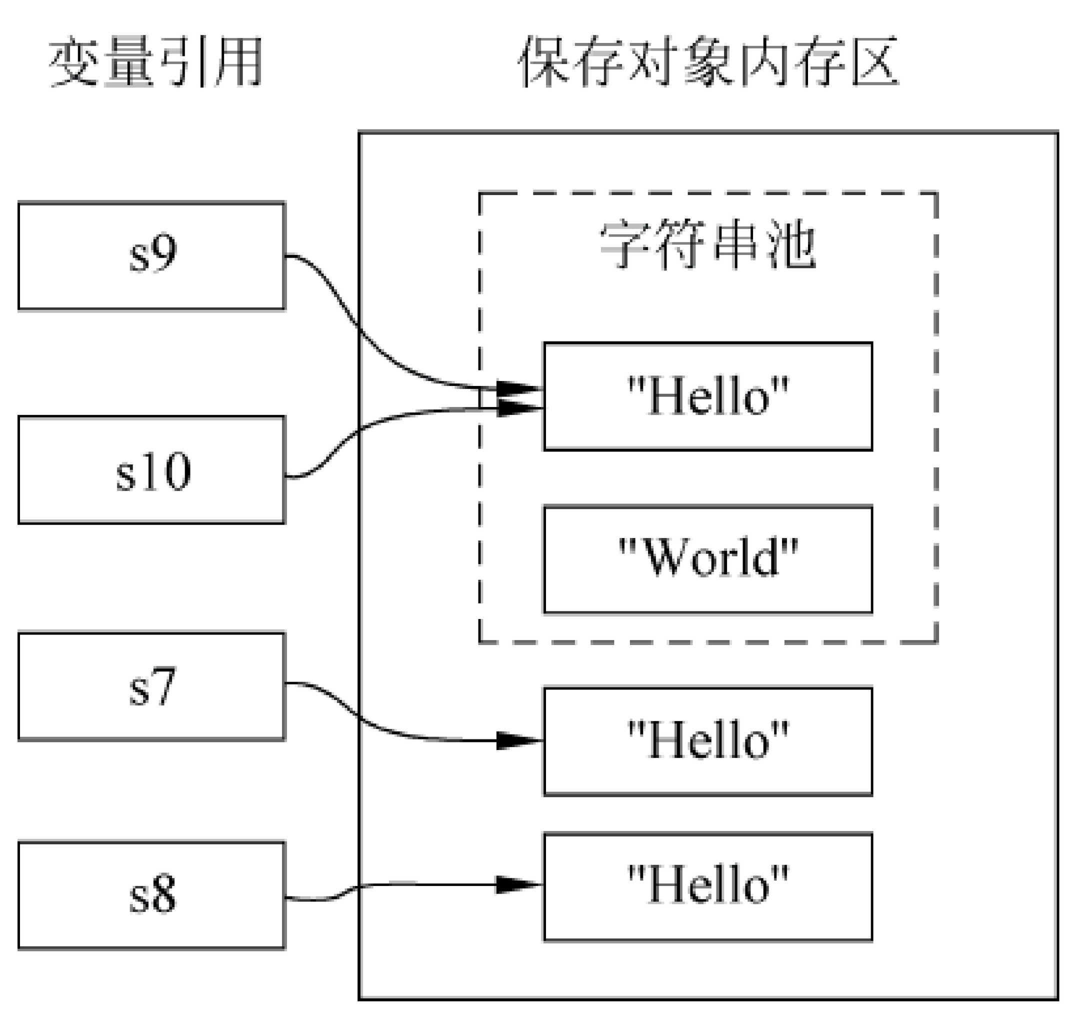


图 9-5　字符串池

9.3.3　字符串拼接

String 字符串虽然是不可变字符串，但也可以进行拼接，只是会产生一个新的对象。String 字符串拼接可以使用+运算符或 String 的 concat(String str)方法。+运算符的优势是可以连接任何类型数据拼接成为字符串，而 concat 方法只能拼接 String 类型字符串。

字符串拼接示例如下：

```
String s1 = "Hello";
//使用+运算符连接
String s2 = s1 + " ";                                  ①
String s3 = s2 + "World";                              ②
System.out.println(s3);

String s4 = "Hello";
//使用+运算符连接,支持+=赋值运算符
s4 += " ";                                             ③
s4 += "World";                                         ④
System.out.println(s4);

String s5 = "Hello";
//使用concat方法连接
s5 = s5.concat(" ").concat("World");                   ⑤
System.out.println(s5);

int age = 18;
String s6 = "她的年龄是" + age + "岁";                  ⑥
System.out.println(s6);

char score = 'A';
String s7 = "她的英语成绩是" + score;                   ⑦
System.out.println(s7);

java.util.Date now = new java.util.Date();             ⑧
//对象拼接自动调用toString()方法
String s8 = "今天是:" + now;                            ⑨
System.out.println(s8);
```

输出结果如下：

```
Hello World
Hello World
Hello World
她的年龄是18岁
她的英语成绩是A
今天是:Thu May 25 16:25:40 CST 2017
```

上述代码第①行和第②行使用＋运算符进行字符串的拼接，其中产生了3个对象。代码第③行和第④行使用＋＝赋值运算符，本质上也是＋运算符进行拼接。

代码第⑤行采用concat方法进行拼接，该方法的完整定义如下：

```
public String concat(String str)
```

它的参数和返回值都是String，因此代码第⑤行可以连续调用该方法进行多个字符串的拼接。

代码第⑥行和第⑦行是使用＋运算符，将字符串与其他类型数据进行拼接。代码第⑨

行是与对象进行拼接，Java 中所有对象都有一个 toString()方法，该方法可以将对象转换为字符串，拼接过程会调用该对象的 toString()方法，将该对象转换为字符串后再进行拼接。代码第⑧行的 java.util.Date 类是 Java SE 提供的日期类。

9.3.4 字符串查找

在给定的字符串中查找字符或字符串是比较常见的操作。在 String 类中提供了 indexOf 和 lastIndexOf 方法用于查找字符或字符串，返回值是查找的字符或字符串所在的位置，−1 表示没有找到。这两个方法有多个重载版本。

- int indexOf(int ch)：从前往后搜索字符 ch，返回第一次找到字符 ch 所在处的索引。
- int indexOf(int ch,int fromIndex)：从指定的索引开始从前往后搜索字符 ch，返回第一次找到字符 ch 所在处的索引。
- int indexOf(String str)：从前往后搜索字符串 str，返回第一次找到字符串 str 所在处的索引。
- int indexOf(String str,int fromIndex)：从指定的索引开始从前往后搜索字符串 str，返回第一次找到字符串 str 所在处的索引。
- int lastIndexOf(int ch)：从后往前搜索字符 ch，返回第一次找到字符 ch 所在处的索引。
- int lastIndexOf(int ch,int fromIndex)：从指定的索引开始从后往前搜索字符 ch，返回第一次找到字符 ch 所在处的索引。
- int lastIndexOf(String str)：从后往前搜索字符串 str，返回第一次找到字符串 str 所在处的索引。
- int lastIndexOf(String str,int fromIndex)：从指定的索引开始从后往前搜索字符串 str，返回第一次找到字符串 str 所在处的索引。

提示 字符串本质上是字符数组，因此它也有索引，索引从零开始。String 的 charAt(int index)方法可以返回索引 index 所在位置的字符。

字符串查找示例代码如下：

```
String sourceStr = "There is a string accessing example.";

//获得字符串长度
int len = sourceStr.length();
//获得索引位置 16 的字符
char ch = sourceStr.charAt(16);

//查找字符和子字符串
int firstChar1 = sourceStr.indexOf('r');
int lastChar1 = sourceStr.lastIndexOf('r');
int firstStr1 = sourceStr.indexOf("ing");
int lastStr1 = sourceStr.lastIndexOf("ing");
int firstChar2 = sourceStr.indexOf('e', 15);
int lastChar2 = sourceStr.lastIndexOf('e', 15);
int firstStr2 = sourceStr.indexOf("ing", 5);
int lastStr2 = sourceStr.lastIndexOf("ing", 5);
```

```
System.out.println("原始字符串:" + sourceStr);
System.out.println("字符串长度:" + len);
System.out.println("索引 16 的字符:" + ch);
System.out.println("从前往后搜索 r 字符,第一次找到它所在索引:" + firstChar1);
System.out.println("从后往前搜索 r 字符,第一次找到它所在索引:" + lastChar1);
System.out.println("从前往后搜索 ing 字符串,第一次找到它所在索引:" + firstStr1);
System.out.println("从后往前搜索 ing 字符串,第一次找到它所在索引:" + lastStr1);
System.out.println("从索引为 15 位置开始,从前往后搜索 e 字符,第一次找到它所在索引:" +
firstChar2);
System.out.println("从索引为 15 位置开始,从后往前搜索 e 字符,第一次找到它所在索引:" +
lastChar2);
System.out.println("从索引为 5 位置开始,从前往后搜索 ing 字符串,第一次找到它所在索引:" +
firstStr2);
System.out.println("从索引为 5 位置开始,从后往前搜索 ing 字符串,第一次找到它所在索引:" +
lastStr2);
```

输出结果如下：

```
原始字符串:There is a string accessing example.
字符串长度:36
索引 16 的字符:g
从前往后搜索 r 字符,第一次找到它所在索引:3
从后往前搜索 r 字符,第一次找到它所在索引:13
从前往后搜索 ing 字符串,第一次找到它所在索引:14
从后往前搜索 ing 字符串,第一次找到它所在索引:24
从索引为 15 位置开始,从前往后搜索 e 字符,第一次找到它所在索引:21
从索引为 15 位置开始,从后往前搜索 e 字符,第一次找到它所在索引:4
从索引为 5 位置开始,从前往后搜索 ing 字符串,第一次找到它所在索引:14
从索引为 5 位置开始,从后往前搜索 ing 字符串,第一次找到它所在索引:-1
```

sourceStr 字符串索引如图 9-6 所示。上述字符串查找方法比较类似，这里重点解释一下 sourceStr.indexOf("ing",5)和 sourceStr.lastIndexOf("ing",5)表达式。从图 9-6 中可见，ing 字符串出现过两次，索引分别是 14 和 24。sourceStr.indexOf("ing",5)表达式从索引为 5 的字符(" ")开始从前往后搜索，结果是找到第一个 ing(索引为 14)，返回值为 14。sourceStr.lastIndexOf("ing",5)表达式从索引为 5 的字符(" ")开始从后往前搜索，没有找到，返回值为－1。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
T	h	e	r	e		i	s		a		s	t	r	i	n	g		a	c	c	e	s	s	i	n	g		e	x	a	m	p	l	e	.

图 9-6 sourceStr 字符串索引

9.3.5 字符串比较

字符串比较是常见的操作，包括比较相等、比较大小、比较前缀和后缀等。

1. 比较相等

String 提供的比较字符串相等的方法如下。

- boolean equals(Object anObject)：比较两个字符串中内容是否相等。

- boolean equalsIgnoreCase(String anotherString)：类似 equals 方法，只是忽略大小写。

2. 比较大小

有时不仅需要知道字符串是否相等，还要知道大小。String 提供的比较字符串大小的方法如下。

- int compareTo(String anotherString)：按字典顺序比较两个字符串。如果参数字符串等于此字符串，则返回值 0；如果此字符串小于参数字符串，则返回一个小于 0 的值；如果此字符串大于参数字符串，则返回一个大于 0 的值。
- int compareToIgnoreCase(String str)：类似 compareTo，只是忽略大小写。

3. 比较前缀和后缀

- boolean endsWith(String suffix)：测试此字符串是否以指定的后缀结束。
- boolean startsWith(String prefix)：测试此字符串是否以指定的前缀开始。

字符串比较示例代码如下：

```
String s1 = new String("Hello");
String s2 = new String("Hello");
//比较字符串是否是相同的引用
System.out.println("s1 == s2 : " + (s1 == s2));
//比较字符串内容是否相等
System.out.println("s1.equals(s2) : " + (s1.equals(s2)));

String s3 = "HELlo";
//忽略大小写比较字符串内容是否相等
System.out.println("s1.equalsIgnoreCase(s3) : " + (s1.equalsIgnoreCase(s3)));

//比较大小
String s4 = "java";
String s5 = "Swift";
//比较字符串大小 s4 > s5
System.out.println("s4.compareTo(s5) : " + (s4.compareTo(s5)));          ①
//忽略大小写比较字符串大小 s4 < s5
System.out.println("s4.compareToIgnoreCase(s5) : " +
                (s4.compareToIgnoreCase(s5)));                          ②

//判断文件夹中文件名
String[] docFolder = { "java.docx", "JavaBean.docx", "Objecitve-C.xlsx", "Swift.docx" };
int wordDocCount = 0;
//查找文件夹中 Word 文档个数
for (String doc : docFolder) {
    //去掉前后空格
    doc = doc.trim();                                                  ③
    //比较后缀是否有.docx 字符串
    if (doc.endsWith(".docx")) {
        wordDocCount++;
    }
}
System.out.println("文件夹中 Word 文档个数是: " + wordDocCount);
```

```
int javaDocCount = 0;
//查找文件夹中 Java 相关文档个数
for (String doc : docFolder) {
    //去掉前后空格
    doc = doc.trim();
    //全部字符转成小写
    doc = doc.toLowerCase();                                              ④
    //比较前缀是否有 java 字符串
    if (doc.startsWith("java")) {
        javaDocCount++;
    }
}
System.out.println("文件夹中 Java 相关文档个数是:" + javaDocCount);
```

输出结果如下：

```
s1 == s2 : false
s1.equals(s2) : true
s1.equalsIgnoreCase(s3) : true
s4.compareTo(s5) : 23
s4.compareToIgnoreCase(s5) : -9
文件夹中 Word 文档个数是: 3
文件夹中 Java 相关文档个数是:2
```

上述代码第①行的 compareTo 方法按字典顺序比较两个字符串，s4.compareTo(s5)表达式返回结果大于 0，说明 s4 大于 s5，字符在字典中的顺序事实上就是它的 Unicode 编码，先比较两个字符串的第一个字符 j 和 S，j 的 Unicode 编码是 106，S 的 Unicode 编码是 83，所以可以得出结论 s4 > s5。代码第②行是忽略大小写时，要么全部当作小写字母进行比较，要么全部当作大写字母进行比较，无论哪种比较，结果都是一样的，即 s4 < s5。

代码第③行 trim()方法可以去除字符串前后空格。代码第④行 toLowerCase()方法可以将此字符串全部转化为小写字符串，类似的方法还有 toUpperCase()方法，可将字符串全部转化为大写字符串。

9.3.6 字符串截取

Java 中字符串 String 截取的主要方法如下。

- String substring(int beginIndex)：从指定索引 beginIndex 开始截取一直到字符串结束的子字符串。
- String substring(int beginIndex，int endIndex)：从指定索引 beginIndex 开始截取直到索引 endIndex－1 处的字符，注意包括索引为 beginIndex 处的字符，但不包括索引为 endIndex 处的字符。

字符串截取方法示例代码如下：

```
String sourceStr = "There is a string accessing example.";
//截取 example.子字符串
String subStr1 = sourceStr.substring(28);             ①
```

```
//截取 string 子字符串
String subStr2 = sourceStr.substring(11, 17);                          ②
System.out.printf("subStr1 = %s%n", subStr1);
System.out.printf("subStr2 = %s%n",subStr2);

//使用 split 方法分隔字符串
System.out.println("-----使用 split 方法-----");
String[] array = sourceStr.split(" ");                                 ③
for (String str : array) {
System.out.println(str);
}
```

输出结果如下：

```
subStr1 = example.
subStr2 = string
-----使用 split 方法-----
There
is
a
string
accessing
example.
```

上述 sourceStr 字符串索引如图 9-6 所示。代码第①行是截取 example. 子字符串，从图 9-6 中可见，e 字符索引是 28，从索引 28 字符截取直到 sourceStr 结尾。代码第②行是截取 string 子字符串，从图 9-6 中可见，s 字符索引是 11，g 字符索引是 16，endIndex 参数应该为 17。

另外，String 还提供了字符串分隔方法，见代码第③行 split(" ")方法，参数是分隔字符串，返回值 String[]。

9.4 可变字符串

可变字符串在追加、删除、修改、插入和拼接等操作过程中不会产生新的对象。

9.4.1 StringBuffer 和 StringBuilder

Java 提供了两个可变字符串类：StringBuffer 和 StringBuilder，中文翻译为“字符串缓冲区”。

StringBuffer 是线程安全的，它的方法是支持线程同步[①]，线程同步会操作串行顺序执行，在单线程环境下会影响效率。StringBuilder 是 StringBuffer 单线程版本，Java 5 之后发布的，它不是线程安全的，但它的执行效率很高。

① 线程同步是一个多线程概念，就是当多个线程访问一个方法时，只能由一个优先级别高的线程先访问，在访问期间会锁定该方法，其他线程只能等到它访问完成释放锁才能访问。有关多线程问题将在后面章节详细介绍。

StringBuffer 和 StringBuilder 具有完全相同的 API，即构造方法和普通方法等内容一样。StringBuilder 中构造方法有 4 个。

- StringBuilder()：创建字符串内容是空的 StringBuilder 对象，初始容量默认为 16 个字符。
- StringBuilder(CharSequence seq)：指定 CharSequence 字符串创建 StringBuilder 对象。CharSequence 接口类型，它的实现类有 String、StringBuffer 和 StringBuilder 等，所以参数 seq 可以是 String、StringBuffer 和 StringBuilder 等类型。
- StringBuilder(int capacity)：创建字符串内容是空的 StringBuilder 对象，初始容量由参数 capacity 指定。
- StringBuilder(String str)：指定 String 字符串创建 StringBuilder 对象。

上述构造方法同样适合于 StringBuffer 类，这里不再赘述。

提示　字符串长度和字符串缓冲区容量的区别：字符串长度是指在字符串缓冲区中目前所包含的字符串长度，通过 length()获得；字符串缓冲区容量是缓冲区中所能容纳的最大字符数，通过 capacity()获得。当所容纳的字符超过这个长度时，字符串缓冲区自动扩充容量，但这是以牺牲性能为代价的扩容。

字符串长度和字符串缓冲区容量示例代码如下：

```
//字符串长度 length 和字符串缓冲区容量 capacity
StringBuilder sbuilder1 = new StringBuilder();
System.out.println("包含的字符串长度:" + sbuilder1.length());
System.out.println("字符串缓冲区容量:" + sbuilder1.capacity());

StringBuilder sbuilder2 = new StringBuilder("Hello");
System.out.println("包含的字符串长度:" + sbuilder2.length());
System.out.println("字符串缓冲区容量:" + sbuilder2.capacity());

//字符串缓冲区初始容量是 16,超过之后会扩容
StringBuilder sbuilder3 = new StringBuilder();
for (int i = 0; i < 17; i++) {
    sbuilder3.append(8);
}
System.out.println("包含的字符串长度:" + sbuilder3.length());
System.out.println("字符串缓冲区容量:" + sbuilder3.capacity());
```

输出结果如下：

```
包含的字符串长度:0
字符串缓冲区容量:16
包含的字符串长度:5
字符串缓冲区容量:21
包含的字符串长度:17
字符串缓冲区容量:34
```

9.4.2 字符串追加

StringBuilder 提供了很多修改字符串缓冲区的方法，如追加、插入、删除和替换等，这一

节先介绍字符串追加方法。字符串追加方法是 append，append 有很多重载方法，可以追加任何类型数据，它的返回值还是 StringBuilder。StringBuilder 的追加法与 StringBuffer 完全一样，这里不再赘述。

字符串追加示例代码如下：

```
//添加字符串、字符
StringBuilder sbuilder1 = new StringBuilder("Hello");          ①
sbuilder1.append(" ").append("World.");                        ②
sbuilder1.append('.');                                         ③
System.out.println(sbuilder1);

StringBuilder sbuilder2 = new StringBuilder();
Object obj = null;
//添加布尔值、转义符和空对象
sbuilder2.append(false).append('\t').append(obj);              ④
System.out.println(sbuilder2);

//添加数值
StringBuilder sbuilder3 = new StringBuilder();
for (int i = 0; i < 10; i++) {
    sbuilder3.append(i);
}
System.out.println(sbuilder3);
```

输出结果如下：

```
Hello World.
false   null
0123456789
```

上述代码第①行是创建一个包含 Hello 字符串的 StringBuilder 对象。代码第②行是两次连续调用 append 方法，由于所有的 append 方法都返回 StringBuilder 对象，所以可以连续调用该方法，这种写法比较简洁。如果不喜欢连续调用 append 方法，则可以将 append 方法占一行，见代码第③行。

代码第④行连续追加了布尔值、转义符和空对象，需要注意的是，布尔值 false 转换为 false 字符串，空对象 null 也转换为"null"字符串。

9.4.3　字符串插入、删除和替换

StringBuilder 中实现插入、删除和替换等操作的常用方法说明如下。

- StringBuilder insert(int offset,String str)：在字符串缓冲区中索引为 offset 的字符位置之前插入 str，insert 有很多重载方法，可以插入任何类型数据。
- StringBuffer delete(int start,int end)：在字符串缓冲区中删除子字符串，要删除的子字符串从指定索引 start 开始直到索引 end−1 处的字符。start 和 end 两个参数与 substring(int beginIndex,int endIndex)方法中的两个参数含义一样。
- StringBuffer replace(int start,int end,String str)：在字符串缓冲区中用 str 替换子

字符串，子字符串从指定索引 start 开始直到索引 end-1 处的字符。start 和 end 同 delete(int start,int end)方法。

以上介绍的虽然是 StringBuilder 方法，但 StringBuffer 也完全一样，这里不再赘述。

示例代码如下：

```
//原始不可变字符串
String str1 = "Java C";
//从不可变的字符串创建可变字符串对象
StringBuilder mstr = new StringBuilder(str1);

//插入字符串
mstr.insert(4, " C++");                          ①
System.out.println(mstr);

//具有追加效果的插入字符串
mstr.insert(mstr.length(), " Objective-C");      ②
System.out.println(mstr);

//追加字符串
mstr.append(" and Swift");
System.out.println(mstr);

//删除字符串
mstr.delete(11, 23);                             ③
System.out.println(mstr);
```

输出结果如下：

```
Java C++ C
Java C++ C Objective-C
Java C++ C Objective-C and Swift
Java C++ C and Swift
```

上述代码第①行 mstr.insert(4," C++")是在索引 4 插入字符串，原始字符串索引如图 9-7 所示，索引 4 位置是一个空格，在它之前插入字符串。代码第②行 mstr.insert(mstr.length(),"Objective-C")是按照字符串的长度插入，也就是在尾部追加字符串。在执行代码第③行删除字符串之前的字符串如图 9-8 所示，mstr.delete(11,23)语句是要删除"Objective-C"子字符串，第一个参数是子字符串开始索引 11；第二个参数是 23，结束字符的索引是 22(end－1)，所以参数 end 是 23。

0	1	2	3	4	5
J	a	v	a		C

图 9-7　原始字符串索引

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
J	a	v	a		C	+	+		C		O	b	j	e	c	t	i	v	e	–	C		a	n	d		S	w	i	f	t

图 9-8　删除之前字符串索引

本章小结

本章介绍了 Java 中的字符串，Java 字符串分为不可变字符串类(String)和可变字符串类(StringBuilder 和 StringBuffer)，并分别介绍了这些字符串类的用法。

9.5 同步练习

1. 下面的语句运行之后 foo 的值是(　　)。

```
String foo = "base";
foo.substring(0, 3);
foo.concat("ket");
foo += "ball";
```

2. 给定如下语句：

```
public class HelloWorld {

    public static void main(String[] args) {

        StringBuffer a = new StringBuffer("A");
        StringBuffer b = new StringBuffer("B");
        operate(a, b);
        System.out.println(a + "," + b);
    }

    static void operate(StringBuffer x, StringBuffer y) {
        y.append(x);
        y = x;

    }
}
```

代码输出结果是(　　)。

A. A,B　　B. A,BA　　C. AB,B　　D. AB,AB

3. 下列代码运行后输出的结果是(　　)。

```
String s1 = "Henry Lee";
String s2 = "Java Applet";
String s3 = "Java";
String st;

if (s1.compareTo(s2) < 0)
    st = s2;
else
    st = s1;
```

```
if (st.compareTo(s3) < 0)
    st = s3;
System.out.println(st);
```

4. 下列代码运行后输出的结果是(　　)。

```
Integer n1 = new Integer(47);
Integer n2 = new Integer(47);
System.out.print(n1 == n2);
System.out.print(",");
System.out.println(n1 != n2);
```

第 10 章 面向对象基础

CHAPTER 10

面向对象是 Java 最重要的特性。Java 是彻底的、纯粹的面向对象语言，在 Java 中“一切都是对象”。本章将介绍面向对象基础知识。

10.1 面向对象简介

面向对象的编程思想：按照真实世界客观事物的自然规律进行分析，客观世界中存在什么样的实体，构建的软件系统就存在什么样的实体。

例如，在真实世界的学校里，会有学生和老师等实体，学生有学号、姓名、所在班级等属性（数据），学生还有学习、提问、吃饭和走路等操作。学生只是抽象的描述，这个抽象的描述称为“类”。在学校里活动的是学生个体，如张同学、李同学等，这些具体的个体称为“对象”，“对象”也称为“实例”。

在现实世界有类和对象，面向对象软件世界也会有，只不过它们会以某种计算机语言编写的程序代码形式存在，这就是面向对象编程（Object Oriented Programming，OOP）。作为面向对象的计算机语言——Java，具有定义类和创建对象等面向对象能力。

10.2 面向对象的三个基本特性

面向对象思想有三个基本特性：封装性、继承性和多态性。

1. 封装性

在现实世界中封装的例子到处都是。例如，一台计算机内部极其复杂，有主板、CPU、硬盘和内存，而一般用户不需要了解它的内部细节，不需要知道主板的型号、CPU 主频、硬盘和内存的大小，于是计算机制造商将用机箱把计算机封装起来，对外提供了一些接口，如鼠标、键盘和显示器等，这样，当用户使用计算机时就变得非常方便。

面向对象的封装与真实世界的目的是一样的。封装能够使外部访问者不能随意存取对象的内部数据，隐藏了对象的内部细节，只保留有限的对外接口。外部访问者不用关心对象的内部细节，使得操作对象变得简单。

2. 继承性

在现实世界中继承也是无处不在的。例如，轮船与客轮之间的关系，客轮是一种特殊轮船，拥有轮船的全部特征和行为，即数据和操作。在面向对象中轮船是一般类，客轮是特殊

类,特殊类拥有一般类的全部数据和操作,称为特殊类继承一般类。在 Java 语言中一般类称为"父类",特殊类称为"子类"。

提示 有些语言如 C++ 支持多继承,多继承就是一个子类可有多个父类,例如,客轮是轮船也是交通工具,客轮的父类是轮船和交通工具。多继承会引起很多冲突问题,因此现在很多面向对象的语言都不支持多继承。Java 语言是单继承的,即只能有一个父类,但 Java 语言可以实现多个接口,可以防止多继承所引起的冲突问题。

3. 多态性

多态性是指在父类中成员变量和成员方法被子类继承之后,可以具有不同的状态或表现行为。有关多态性详细解释可参考 12.4 节,这里不再赘述。

10.3 类

类是 Java 语言中的一种重要的引用数据类型,是组成 Java 程序的基本要素。它封装了一类对象的数据和操作。

10.3.1 类声明

Java 语言中一个类的实现包括类声明和类体。类声明语法格式如下:

```
[public][abstract|final] class className [extends superclassName] [implements interfaceNameList] {
    //类体
}
```

其中,class 是声明类的关键字,className 是自定义的类名;class 前面的修饰符 public、abstract、final 用来声明类,它们可以省略,其具体用法后面章节会详细介绍;superclassName 为父类名,可以省略,如果省略,则该类继承 Object 类,Object 类为所有类的根类,所有类都直接或间接继承 Object;interfaceNameList 是该类实现的接口列表,可以省略,接口列表中的多个接口之间用逗号分隔。

提示 本书语法表示符号约定,在语法说明中,括号([])表示可以省略;竖线(|)表示"或关系",如 abstract|final,说明可以使用 abstract 或 final 关键字,两个关键字不能同时出现。

声明动物(Animal)类代码如下:

```
//Animal.java
public class Animal extends Object {

    //类体
}
```

上述代码声明了动物类,它继承了 Object 类。继承 Object 类 extends Object 代码可以省略。

类体是类的主体,包括数据和操作,即成员变量和成员方法。下面展开介绍一下。

10.3.2　成员变量

声明类体中成员变量语法格式如下：

```
class className {
    [public | protected | private ] [static] [final] type variableName;     //成员变量
}
```

其中，type是成员变量数据类型，variableName是成员变量名。type前的关键字都是成员变量修饰符，说明如下。

(1) public、protected和private修饰符用于封装成员变量。

(2) static修饰符用于声明静态变量，所以静态变量也称为"类变量"。

(3) final修饰符用于声明变量，该变量不能被修改。

下面看一个声明成员变量示例：

```
//Animal.java
public class Animal extends Object {

    //动物年龄
    int age = 1;
    //动物性别
    public boolean sex = false;
    //动物体重
    private double weight = 0.0;

}
```

上述代码中没有展示静态变量声明，有关静态变量稍后会详细介绍。

10.3.3　成员方法

声明类体中成员方法语法格式如下：

```
class className {

     [public | protected | private ] [static] [final | abstract] [native] [synchronized]
            type methodName([paramList]) [throws exceptionList] {
                //方法体
    }
}
```

其中，type是方法返回值数据类型，methodName是方法名。type前的关键字都是方法修饰符，说明如下。

(1) public、protected和private修饰符用于封装方法。

(2) static修饰符用于声明静态方法，所以静态方法也称为"类方法"。

(3) final | abstract不能同时修饰方法，final修饰的方法不能在子类中被覆盖；abstract用来修饰抽象方法，抽象方法必须在子类中被实现。

（4）native 修饰的方法，称为“本地方法”，本地方法调用平台本地代码（如 C 或 C++编写的代码），不能实现跨平台。

（5）synchronized 修饰的方法是同步的，当多线程方式同步方法时，只能串行地执行，保证线程是安全的。

方法声明中还有（[paramList]）部分，它是方法的参数列表。[throws exceptionList]是声明抛出异常列表。

下面看一个声明方法示例：

```
public class Animal {//extends Object {

    //动物年龄
    int age = 1;
    //动物性别
    public boolean sex = false;
    //动物体重
    private double weight = 0.0;

    public void eat() {                                             ①
        //方法体
        return;                                                     ②
    }

    int run() {                                                     ③
        //方法体
        return 10;                                                  ④
    }

    protected int getMaxNumber(int number1, int number2) {          ⑤
        //方法体
        if (number1 > number2) {
            return number1;                                         ⑥
        }
        return number2;
    }
}
```

上述代码第①、③、⑤行声明了 3 个方法。方法在执行完毕后把结果返还给它的调用者，方法体包含“return 返回结果值；”语句，见代码第④行的“return 10；”，“返回结果值”数据类型与方法的返回值类型要匹配。如果方法返回值类型为 void 时，方法体包含“return；”语句，见代码第②行，如果“return；”语句是最后一行，则可以省略。

提示 return 语句通常用在一个方法体的最后，否则会产生编译错误，除非用在 if-else 语句中，见代码第⑥行。

10.4 包

在程序代码中为类起一个名字是非常重要的，但是有时会出现非常尴尬的情况，名字会发生冲突。例如，项目中自定义了一个日期类，为它取名为 Date，但是会发现 Java SE 核心

库中还有两个 Date，它们分别位于 java.util 包和 java.sql 包中。

10.4.1 包作用

在 Java 中为了防止类、接口、枚举和注释等命名冲突引用了包(package)概念，包本质上是命名空间(namespace)[①]。在包中可以定义一组相关的类型(类、接口、枚举和注释)，并为它们提供访问保护和命名空间管理。

在前面提到的 Date 类名称冲突问题很好解决，将不同 Date 类放到不同的包中，自定义 Date，可以放到自己定义的包 com.a51work6 中，这样就不会与 java.util 包和 java.sql 包中的 Date 发生冲突问题。

10.4.2 包定义

Java 中使用 package 语句定义包，package 语句应该放在源文件的第一行，在每个源文件中只能有一个包定义语句，并且 package 语句适用于所有类型(类、接口、枚举和注释)的文件。定义包语法格式如下：

```
package pkg1[.pkg2[.pkg3…]];
```

pkg1～pkg3 都是组成包名的一部分，之间用点(.)连接。首先它们的命名应该是合法的标识符；其次应该遵守 Java 包命名规范，即全部是小写字母。

定义包示例代码如下：

```
// Date.java 文件
package com.a51work6;

public class Date {

}
```

com.a51work6 是自定义的包名，包名一般是公司域名的倒置。

提示 笔者公司的域名是 51work6.com，倒置后是 com.51work6，其中 51work6 是非法标识符(不能用数字开头)，所以 com.51work6 包名是非法的，于是将包名改为 com.a51work6。

如果在源文件中没有定义包，那么类、接口、枚举和注释类型文件将会被放进一个无名的包中，也称为默认包。

定义好包后，包采用层次结构管理这些类型(类、接口、枚举和注释)，如图 10-1 所示是在 Eclipse 包资源视图中查看包，可见有默认包和 com.a51work6 包。如果在文件系统中查看这些包，会发现如图 10-2 所示的层次结构，源文件目录是根目录，也是默认包目录，可见其中有一个 HelloWorld.java 文件。com 是文件夹，a51work6 是子文件夹，在 a51work6 中

---

① 命名空间，也称名字空间、名称空间等，它表示着一个标识符(identifier)的可见范围。一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。这样，在一个新的命名空间中可定义任何标识符，它们不会与任何已有的标识符发生冲突，因为已有的定义都处于其他命名空间中。

——引自于维基百科 https://zh.wikipedia.org/wiki/命名空间

包含 Animal.java 和 Date.java 两个文件。Java 编译器把包对应于文件系统的目录管理，不仅是源文件，编译之后的字节码文件也是采用文件系统的目录管理的。



图 10-1　在 Eclipse 包资源视图中查看包

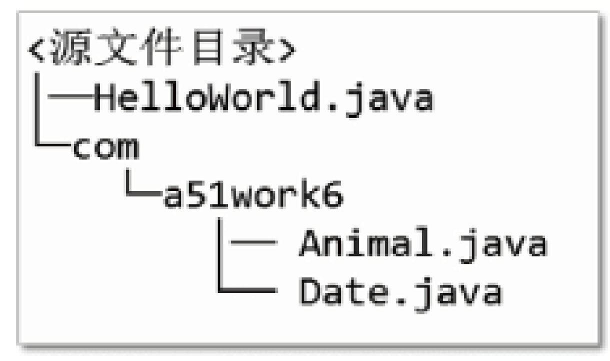


图 10-2　文件系统目录与包

10.4.3　包引入

为了能够使用一个包中类型(类、接口、枚举和注释)，需要在 Java 程序中明确引入该包。使用 import 语句实现引入包，import 语句应位于 package 语句之后，所有类的定义之前，可以有 0～n 条 import 语句，其语法格式为：

```
import package1[.package2…].(类型名|*);
```

“包名.类型名”形式只引入具体类型，“包名.*”采用通配符，表示引入这个包下所有的类型。但从编程规范的角度提倡明确引入类型名，即“包名.类型名”形式可以提高程序的可读性。

如果需要在程序代码中使用 com.a51work6 包中 Date 类，示例代码如下：

```
//HelloWorld.java 文件
import com.a51work6.Date;                                    ①

public class HelloWorld {

    public static void main(String[] args) {

        Date date = new Date();                              ②
        System.out.println(date);
    }
}
```

上述代码第②行使用了 Date 类，需要引入 Date 所在的包，见代码第①行，import 是关键字，代码第①行的 import 语句采用“包名.类型名”形式。

如果在一个源文件中引入两个相同包名+类型名，见如下代码，代码第②行会发生编译错误。为避免这个编译错误，可以在没有引入包的类型名前加上包名，详见如下代码第②行中的 java.util.Date：

```
//HelloWorld.java 文件
import com.a51work6.Date;
```

```
//import java.util.Date;                              ①

public class HelloWorld {

    public static void main(String[] args) {

        Date date = new Date();
        System.out.println(date);
        java.util.Date now = new java.util.Date();     ②
        System.out.println(now);

    }
}
```

注意　当前源文件与要使用的类型(类、接口、枚举和注释)在同一个包中,可以不用引入包。

10.4.4　常用包

Java SE 提供了一些常用包,其中包含 Java 开发中常用的基础类。这些包有java.lang、java.io、java.net、java.util、java.text、java.awt 和 javax.swing。

1. java.lang 包

java.lang 包中包含了 Java 语言的核心类,如 Object、Class、String、包装类和 Math 等,还有包装类 Boolean、Character、Integer、Long、Float 和 Double。使用 java.lang 包中的类型,不需要显式使用 import 语句引入,它是由解释器自动引入。

2. java.io 包

java.io 包中提供多种输入输出流类,如 InputStream、OutputStream、Reader 和 Writer。还有文件管理相关类和接口,如 File 和 FileDescriptor 类以及 FileFilter 接口。

3. java.net 包

java.net 包中包含进行网络相关操作的类,如 URL、Socket 和 ServerSocket 等。

4. java.util 包

java.util 包中包含一些实用工具类和接口,如集合、日期和日历相关类及接口。

5. java.text 包

java.text 包中提供文本处理、日期格式化和数字格式化等相关类及接口。

6. java.awt 和 javax.swing 包

java.awt 和 javax.swing 包提供了 Java 图形用户界面开发所需要的各种类和接口。java.awt 提供了一些基础类和接口,javax.swing 提供了一些高级组件。

10.5　方法重载

在第 9 章介绍字符串时就已经用到过方法重载(overload),这一节详细介绍一下重载。出于使用方便等原因,在设计一个类时将具有相似功能的方法起相同的名字。例如,String 字符串查找方法 indexOf 有很多不同版本,如图 10-3 所示。

int	indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引。
int	indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
int	indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引。
int	indexOf(String str, int fromIndex) 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。

图 10-3　indexOf 方法重载

这些相同名字的方法之所以能够在一个类中同时存在，是因为它们的方法参数列表，调用时根据参数列表调用相应重载方法。

提示　方法重载中参数列表不同的含义是参数的个数不同或者参数类型不同。另外，返回类型不能用来区分方法重载。

方法重载示例 MethodOverloading.java 代码如下：

```
//MethodOverloading.java 文件
package com.a51work6;

class MethodOverloading {

    void receive(int i) {                                              ①
        System.out.println("接收一个 int 参数");
        System.out.println("i = " + i);
    }

    void receive(int x, int y) {                                       ②
        System.out.println("接收两个 int 参数");
        System.out.printf("x = %d, y = %d \r", x, y);
    }

    int receive(double x, double y) {                                  ③
        System.out.println("接收两个 double 参数");
        System.out.printf("x = %f, y = %f \r", x, y);
        return 0;
    }
}

//HelloWorld.java 文件调用 MethodOverloading
package com.a51work6;

public class HelloWorld {
    public static void main(String[] args) {

        MethodOverloading mo = new MethodOverloading();

        //调用 void receive(int i)
```

```
            mo.receive(1);                                  ④

            //调用 void receive(int x, int y)
            mo.receive(2, 3);                               ⑤

            //调用 void receive(double x, double y)
            mo.receive(2.0, 3.3);                           ⑥
        }
    }
```

MethodOverloading 类中有 3 个相同名字的 receive 方法，在 HelloWorld 的 main 方法中调用 MethodOverloading 的 receive 方法。运行结果如下：

```
接收一个 int 参数
i = 1
接收两个 int 参数
x = 2, y = 3
接收两个 double 参数
x = 2.000000, y = 3.300000
```

调用哪一个 receive 方法是根据参数列表决定的。如果参数类型不一致，编译器会进行自动类型转换，寻找适合版本的方法，如果没有适合方法，则会发生编译错误。假设删除代码第②行的 void receive(int x,int y)方法，代码第⑤行的 mo.receive(2,3)语句调用的是 void receive(double x,double y)方法，其中 int 类型参数(2 和 3)会自动转换为 double 类型(2.0 和 3.0)再调用。

10.6 封装性与访问控制

Java 面向对象的封装性是通过对成员变量和方法进行访问控制实现的，访问控制分为 4 个等级：私有、默认、保护和公有，具体规则如表 10-1 所示。

表 10-1 Java 类成员的访问控制

控制等级	同一个类	同一个包	不同包的子类	不同包非子类
私有	Yes			
默认	Yes	Yes		
保护	Yes	Yes	Yes	
公有	Yes	Yes	Yes	Yes

下面详细解释一下这 4 种访问级别。

10.6.1 私有级别

私有级别的关键字是 private，私有级别的成员变量和方法只能在其所在类的内部自由使用，在其他的类中则不允许直接访问。私有级别限制性最高。私有级别示例代码如下：

```
//PrivateClass.java 文件
package com.a51work6;

public class PrivateClass {                        ①

    private int x;                                 ②

    public PrivateClass() {                        ③
        x = 100;
    }

    private void printX() {                        ④
        System.out.println("Value Of x is" + x);
    }

}

//HelloWorld.java 文件调用 PrivateClass
package com.a51work6;

public class HelloWorld {
    public static void main(String[] args) {

        PrivateClass p;
        p = new PrivateClass();

        //编译错误,PrivateClass 中的方法 printX()不可见
        p.printX();                                ⑤
    }
}
```

上述代码第①行声明 PrivateClass 类,其中的代码第②行是声明私有实例变量 x,代码第③行是声明公有的构造方法,构造方法将在第 11 章详细介绍。代码第④行声明私有实例方法。

HelloWorld 类中代码第⑤行会有编译错误,因为 PrivateClass 中 printX()的方法是私有方法。

10.6.2 默认级别

默认级别没有关键字,也就是没有访问修饰符,默认级别的成员变量和方法可以在其所在类内部和同一个包的其他类中被直接访问,但在不同包的类中则不允许直接访问。

默认级别示例代码如下:

```
//DefaultClass.java 文件
package com.a51work6;

public class DefaultClass {
```

```
    int x;                                    ①

    public DefaultClass() {
        x = 100;
    }

    void printX() {                           ②
        System.out.println("Value Of x is" + x);
    }

}
```

上述代码第①行的 x 变量前没有访问限制修饰符，代码第②行的方法也没有访问限制修饰符。它们的访问级别都有默认访问级别。

在相同包(com.a51work6)中调用 DefaultClass 类代码如下：

```
//com.a51work6 包中 HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        DefaultClass p;
        p = new DefaultClass();
        p.printX();
    }
}
```

默认访问级别可以在同一包中访问，上述代码可以编译通过。

在不同的包中调用 DefaultClass 类代码如下：

```
//默认包中 HelloWorld.java 文件
import com.a51work6.DefaultClass;

public class HelloWorld {

    public static void main(String[] args) {

        DefaultClass p;
        p = new DefaultClass();
        //编译错误,DefaultClass 中的方法 printX()不可见
        p.printX();
    }
}
```

该 HelloWorld.java 文件与 DefaultClass 类不在同一个包中，printX()是默认访问级别，所以 p.printX()方法无法编译通过。

10.6.3 保护级别

保护级别的关键字是 protected，保护级别在同一包中完全与默认访问级别一样，但是不同包中子类能够继承父类中的 protected 变量和方法，这就是所谓的保护级别，“保护”就是保护某个类的子类都能继承该类的变量和方法。

保护级别示例代码如下：

```
//ProtectedClass.java 文件
package com.a51work6;

public class ProtectedClass {

    protected int x;                              ①

    public ProtectedClass() {
        x = 100;
    }

    protected void printX() {                     ②
        System.out.println("Value Of x is " + x);
    }

}
```

上述代码第①行的 x 变量是保护级别，代码第②行的方法也是保护级别。

在相同包(com.a51work6)中调用 ProtectedClass 类代码如下：

```
//默认包中 HelloWorld.java 文件
package com.a51work6;

import com.a51work6.ProtectedClass;

public class HelloWorld {

    public static void main(String[] args) {

        ProtectedClass p;
        p = new ProtectedClass();
        //同一包中可以直接访问 ProtectedClass 中的方法 printX()
        p.printX();

    }
}
```

同一包中保护访问级别与默认访问级别一样，可以直接访问 ProtectedClass 的 printX()方法，上述代码可以编译通过。

在不同的包中调用 ProtectedClass 类代码如下：

```
//默认包中 HelloWorld.java 文件
import com.a51work6.ProtectedClass;

public class HelloWorld {

    public static void main(String[] args) {

        ProtectedClass p;
        p = new ProtectedClass();
        //编译错误,不同包中不能直接访问保护方法 printX()
        p.printX();

    }
}
```

该 HelloWorld.java 文件与 ProtectedClass 类不在同一个包中，不同包中不能直接访问保护方法 printX()，所以 p.printX()方法无法编译通过。

在不同的包中继承 ProtectedClass 类代码如下：

```
//默认包中 SubClass.java 文件
import com.a51work6.ProtectedClass;

public class SubClass extends ProtectedClass {

    void display() {
        //printX()方法是从父类继承过来
        printX();                          ①
        //x 实例变量是从父类继承过来
        System.out.println(x);             ②
    }
}
```

不同包中 SubClass 从 ProtectedClass 类继承了 printX()方法和 x 实例变量。代码第①行是调用从父类继承下来的方法，代码第②行是调用从父类继承下来的实例变量。

提示　访问成员有两种方式：一种是调用，即通过类或对象调用它的成员，如 p.printX()语句；另一种是继承，即子类继承父类的成员变量和方法。

- 公有访问级别在任何情况下两种方式都可以。
- 默认访问级别在同一包中两种访问方式都可以，不能在包之外访问。
- 保护访问级别在同一包中与默认访问级别一样，两种访问方式都可以，但是在不同包之外只能继承访问。
- 私有访问级别只能在本类中通过调用方法访问，不能继承访问。

提示　访问类成员时，在能满足使用的前提下，应尽量限制类中成员的访问级别，访问级别顺序是：私有级别→默认级别→保护级别→公有级别。

10.6.4 公有级别

公有级别的关键字是 public，公有级别的成员变量和方法可以在任何场合被直接访问，是最宽松的一种访问控制等级。

公有级别示例代码如下：

```
//PublicClass.java 文件
package com.a51work6;

public class PublicClass {

    public int x;                                    ①

    public PublicClass() {
        x = 100;
    }

    public void printX() {                           ②
        System.out.println("Value Of x is" + x);
    }

}
```

上述代码第①行的 x 变量是公有级别，代码第②行的方法也是公有级别。调用 PublicClass 类代码如下：

```
//默认包中 HelloWorld.java 文件
import com.a51work6.PublicClass;

public class HelloWorld {

    public static void main(String[] args) {

        PublicClass p;
        p = new PublicClass();
        p.printX();
    }
}
```

该 HelloWorld.java 文件与 PublicClass 类不在同一个包中，可以直接访问公有的 printX()方法。

10.7 静态变量和静态方法

有一个 Account(银行账户)类，假设它有 3 个成员变量：amount(账户金额)、interestRate(利率)和 owner(账户名)。在这 3 个成员变量中，amount 和 owner 会因人而异，对于不同的账户这些内容是不同的，而所有账户的 interestRate 都是相同的。

amount 和 owner 成员变量与账户个体有关，称为"实例变量"，interestRate 成员变量与个体无关，或者说是所有账户个体共享的，这种变量称为"静态变量"或"类变量"。

静态变量和静态方法示例代码如下：

```
//Account.java 文件
package com.a51work6;

public class Account {

    //实例变量账户金额
    double amount = 0.0;                                    ①
    //实例变量账户名
    String owner;                                           ②

    //静态变量利率
    static double interestRate = 0.0668;                    ③

    //静态方法
    public static double interestBy(double amt) {          ④
        //静态方法可以访问静态变量和其他静态方法
        return interestRate * amt;                          ⑤
    }

    //实例方法
    public String messageWith(double amt) {                 ⑥
        //实例方法可以访问实例变量、实例方法、静态变量和静态方法
        double interest = Account.interestBy(amt);          ⑦
        StringBuilder sb = new StringBuilder();
        //拼接字符串
        sb.append(owner).append("的利息是").append(interest);
        //返回字符串
        return sb.toString();
    }
}
```

static 修饰的成员变量是静态变量，见代码第③行，static 修饰的方法是静态方法，见代码第④行。相反，没有 static 修饰的成员变量是实例变量，见代码第①行和第②行，没有 static 修饰的方法是实例方法，见代码第⑥行。

注意　**静态方法可以访问静态变量和其他静态方法，例如访问代码第⑤行中的 interestRate 静态变量。实例方法可以访问实例变量、其他实例方法、静态变量和静态方法，例如访问代码第⑦行中的 interestBy 静态方法。**

调用 Account 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {
```

```
    public static void main(String[ ] args) {
        //访问静态变量
        System.out.println(Account.interestRate);               ①
        //访问静态方法
        System.out.println(Account.interestBy(1000));           ②

        Account myAccount = new Account();
        //访问实例变量
        myAccount.amount = 1000000;                             ③
        myAccount.owner = "Tony";                               ④
        //访问实例方法
        System.out.println(myAccount.messageWith(1000));        ⑤

        //通过实例访问静态变量
        System.out.println(myAccount.interestRate);             ⑥
    }
}
```

调用静态变量或静态方法时，可以通过类名或实例名调用，代码第①行中的 Account.interestRate 通过类名调用静态变量，代码第②行中的 Account.interestBy(1000)是通过类名调用静态方法，代码第⑥行是通过实例调用静态变量。

10.8 静态代码块

前面介绍的静态变量 interestRate 可以在声明同时初始化，如下代码所示：

```
public class Account {

    //静态变量利率
    static double interestRate = 0.0668;
    …
}
```

如果初始化静态变量不是简单常量，则需要进行计算才能初始化，可以使用静态(static)代码块，静态代码块在类第一次加载时执行，并只执行一次。示例代码如下：

```
//Account.java 文件
package com.a51work6;

public class Account {

    //实例变量账户金额
    double amount = 0.0;
    //实例变量账户名
    String owner;

    //静态变量利率
    static double interestRate;
```

```
    //静态方法
    public static double interestBy(double amt) {
        //静态方法可以访问静态变量和其他静态方法
        return interestRate * amt;
    }

    //静态代码块
    static {                                                    ①
        System.out.println("静态代码块被调用…");
        //初始化静态变量
        interestRate = 0.0668;                                  ②
    }
}
```

上述代码第①行是静态代码块，在静态代码块中可以初始化静态变量，见代码第②行，也可以调用静态方法。

调用 Account 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Account myAccount = new Account();                          ①
        //访问静态变量
        System.out.println(Account.interestRate);                   ②
        //访问静态方法
        System.out.println(Account.interestBy(1000));

    }
}
```

Account 静态代码块是在第一次加载 Account 类时调用。上述代码第①行是第一次使用 Account 类，此时会调用静态代码块。

本章小结

本章主要介绍了面向对象基础知识。首先介绍了面向对象的一些基本概念、面向对象的三个基本特性。然后介绍了类、包、方法重载和访问控制。最后介绍了静态变量、静态方法和静态代码块。

10.9 同步练习

1. 下列哪一项不属于面向对象程序设计的基本要素？（　　）

A. 类　　B. 对象　　C. 方法　　D. 安全

2. 定义一个类名为"MyClass.java"的类,并且该类可被一个项目中的所有类访问,那么该类的正确声明应为(　　)。

A. private class MyClass extends Object　　B. class MyClass extends Object

C. public class MyClass　　D. public class MyClass extends Object

3. 判断对错:用 static 修饰的方法称为静态方法,它不属于类的一个具体对象,而是整个类的类方法。(　　)

4. 下面哪项编译不会有错?(　　)

A.
```
package testpackage;
public class Test{//do something…}
class MyClass{}
```

B.
```
import java.io.*;
package testpackage;
public class Test{//do something…}
```

C.
```
import java.io.*;
class Person{//do something…}
public class Test{//do something…}
```

D.
```
import java.io.*;
import java.awt.*;
public class Test{//do something…}
```

5. 下述哪些说法是正确的?(　　)

A. 实例变量是类的成员变量　　B. 实例变量是用 static 关键字声明的

C. 方法变量在方法执行时创建　　D. 方法变量在使用之前必须初始化

6. 指出(　　)与方法 public void add(int a){}为合理的重载方法。

A. public int add(int a)　　B. public void add(long a)

C. public void add(int a,int b)　　D. public void add(float a)

第 11 章

CHAPTER 11

对　　象

类实例化可生成对象，实例方法就是对象方法，实例变量就是对象属性。一个对象的生命周期包括 3 个阶段：创建、使用和销毁。前面章节已经多次用到了对象，本章详细介绍一下对象的创建和销毁等相关知识。

11.1　创建对象

创建对象包括两个步骤：声明和实例化。

1. 声明

声明对象与声明普通变量没有区别，语法格式如下：

```
type objectName;
```

其中，type 是引用类型，即类、接口和数组。示例代码如下：

```
String name;
```

该语句声明了字符串类型对象 name。可以声明并不为对象分配内存空间，而只是分配一个引用。

2. 实例化

实例化过程分为两个阶段：为对象分配内存空间和初始化对象，首先使用 new 运算符为对象分配内存空间，然后再调用构造方法初始化对象。示例代码如下：

```
String name;
name = new String("Hello World");
```

代码中 String("Hello World")表达式就是调用 String 的构造方法。初始化完成之后如图 11-1 所示。

11.2　空对象

一个引用变量没有通过 new 分配内存空间，这个对象就是空对象。Java 使用关键字 null 表示空对象。示例代码如下：

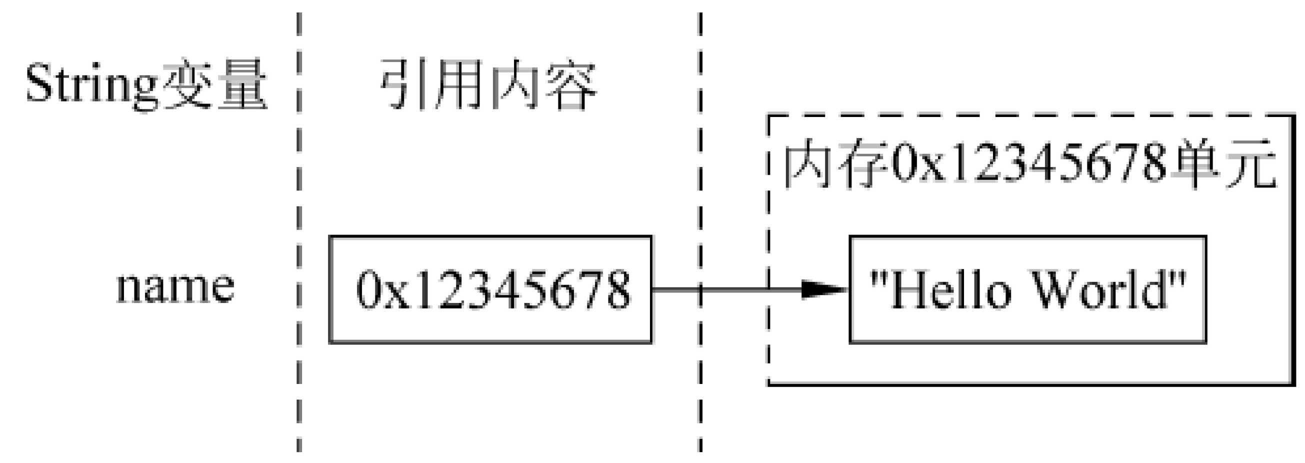


图 11-1　对象实例化

```
String name = null;
name = "Hello World";
```

引用变量默认值是 null。当试图调用一个空对象的实例变量或实例方法时，会抛出空指针异常 NullPointerException，代码如下所示：

```
String name = null;
//输出 null 字符串
System.out.println(name);
//调用 length()方法
int len = name.length();                ①
```

但是代码运行到第①行时，系统会抛出异常。这是因为调用 length()方法时，name 是空对象。程序员应该避免调用空对象的成员变量和方法，代码如下：

```
//判断对象是否为 null
if (name != null) {
    int len = name.length();
}
```

提示　产生空对象有两种可能性：第一是程序员自己忘记了实例化；第二是空对象是别人传递过来的。程序员必须防止第一种情况发生，应该仔细检查自己的代码，为自己创建的所有对象进行实例化并初始化。第二种情况需要通过判断对象非 null 进行避免。

11.3　构造方法

在 11.1 节使用了表达式 new String("Hello World")，其中 String("Hello World")是调用构造方法。构造方法是类中特殊方法，用来初始化类的实例变量，这个就是构造方法，它在创建对象(new 运算符)之后自动调用。

Java 构造方法的特点如下。

(1) 构造方法名必须与类名相同。

(2) 构造方法没有任何返回值，包括 void。

(3) 构造方法只能与 new 运算符结合使用。

构造方法示例代码如下：

```
//Rectangle.java 文件
package com.a51work6;
```

```
//矩形类
public class Rectangle {

    //矩形宽度
    int width;
    //矩形高度
    int height;
    //矩形面积
    int area;

    //构造方法
    public Rectangle(int w, int h) {            ①
        width = w;
        height = h;
        area = getArea(w, h);
    }
    …
}
```

代码第①行声明了一个构造方法，其中有两个参数 w 和 h，用来初始化 Rectangle 对象的两个成员变量 width 和 height，注意前面没有任何的返回值。

11.3.1 默认构造方法

有时在类中根本看不到任何构造方法。例如本节中 User 类代码如下：

```
//User.java 文件
package com.a51work6;

public class User {

    //用户名
    private String username;

    //用户密码
    private String password;

}
```

从上述 User 类代码(只有两个成员变量)中看不到任何构造方法，但是还是可以调用无参数的构造方法创建 User 对象，见如下代码：

```
//HelloWorld.java 文件
…
User user = new User();
```

Java 虚拟机为没有构造方法的类提供一个无参数的默认构造方法，默认构造方法其方法体内无任何语句，默认构造方法相当于如下代码：

```
//默认构造方法
public User() {
}
```

默认构造方法的方法体内无任何语句，也就不能够初始化成员变量了。那么这些成员变量就会使用默认值，成员变量默认值与数据类型有关。具体内容可以参考 8.1.2 节中的表 8-1，这里不再赘述。

11.3.2 构造方法重载

在一个类中可以有多个构造方法，它们具有相同的名字（与类名相同），参数列表不同，所以它们之间一定是重载关系。

构造方法重载示例代码如下：

```
//Person.java 文件
package com.a51work6;

import java.util.Date;

public class Person {

    //名字
    private String name;
    //年龄
    private int age;
    //出生日期
    private Date birthDate;

    public Person(String n, int a, Date d) {            ①
        name = n;
        age = a;
        birthDate = d;
    }

    public Person(String n, int a) {                    ②
        name = n;
        age = a;
    }

    public Person(String n, Date d) {                   ③
        name = n;
        birthDate = d;
    }

    public Person(String n) {                           ④
        name = n;
    }

    public String getInfo() {
```

```
            StringBuilder sb = new StringBuilder();
            sb.append("名字: ").append(name).append('\n');
            sb.append("年龄: ").append(age).append('\n');
            sb.append("出生日期: ").append(birthDate). append('\n');
            return sb.toString();
        }
    }
```

上述代码 Person 类代码提供了 4 个重载的构造方法，如果有准确的姓名、年龄和出生日期信息，则可以选用代码第①行的构造方法创建 Person 对象；如果只有姓名和年龄信息，则可选用代码第②行的构造方法创建 Person 对象；如果只有姓名和出生日期信息，则可选用代码第③行的构造方法创建 Person 对象；如果只有姓名信息，则可选用代码第④行的构造方法创建 Person 对象。

11.3.3 构造方法封装

构造方法也可以进行封装，访问级别与普通方法一样，构造方法的访问级别参考表 11-1。示例代码如下：

```
//Person.java 文件
package com.a51work6;

import java.util.Date;

public class Person {

    //名字
    private String name;
    //年龄
    private int age;
    //出生日期
    private Date birthDate;

    //公有级别限制
    public Person(String n, int a, Date d) {       ①
        name = n;
        age = a;
        birthDate = d;
    }

    //默认级别限制
    Person(String n, int a) {                      ②
        name = n;
        age = a;
    }

    //保护级别限制
    protected Person(String n, Date d) {           ③
```

```
        name = n;
        age = 30;
        birthDate = d;
    }

    //私有级别限制
    private Person(String n) {                    ④
        name = n;
        age = 30;
    }

    …
}
```

上述代码第①行是声明公有级别的构造方法。代码第②行是声明默认级别，默认级别只能在同一个包中访问。代码第③行是保护级别的构造方法，该构造方法在同一个包中与默认级别相同，在不同包中可以被子类继承。代码第④行是私有级别构造方法，该构造方法只能在当前类中使用，不允许在外边访问，私有构造方法可以应用于单例设计模式[①]等设计。

11.4 this 关键字

前面章节中使用过 this 关键字，this 指向对象本身，一个类可以通过 this 来获得一个代表它自身的对象变量。this 使用在如下 3 种情况中。

- 调用实例变量。
- 调用实例方法。
- 调用其他构造方法。

使用 this 变量的示例代码如下：

```
//Person.java 文件
package com.a51work6;

import java.util.Date;

public class Person {

    //名字
    private String name;
    //年龄
    private int age;
    //出生日期
    private Date birthDate;
```

① 单例设计模式是一种常用的软件设计模式，其可以保证系统中一个类只有一个实例。

```
    //3 个参数构造方法
    public Person(String name, int age, Date d) {          ①
        this.name = name;                                  ②
        this.age = age;                                    ③
        birthDate = d;
        System.out.println(this.toString());               ④
    }

    public Person(String name, int age) {
        //调用 3 个参数构造方法
        this(name, age, null);                             ⑤
    }

    public Person(String name, Date d) {
        //调用 3 个参数构造方法
        this(name, 30, d);                                 ⑥
    }

    public Person(String name) {
        //System.out.println(this.toString());
        //调用 Person(String name, Date d)构造方法
        this(name, null);                                  ⑦
    }

    @Override
    public String toString() {
        return "Person [name=" + name                      ⑧
                + ", age=" + age                           ⑨
                + ", birthDate=" + birthDate + "]";
    }
}
```

上述代码中多次用到了 this 关键字，下面详细分析一下。代码第①行声明 3 个参数构造方法，其中参数 name 和 age 与实例变量 name 和 age 命名冲突，参数是作用域为整个方法的局部变量，为了防止局部变量与成员变量命名发生冲突，可以使用 this 调用成员变量，见代码第②行和第③行。注意代码第⑧行和第⑨行的 name 和 age 变量没有冲突，所以可以不使用 this 调用。

this 也可以调用本对象的方法，见代码第④行的 this.toString()语句，在本例中 this 可以省略。

在多个构造方法重载时，一个构造方法可以调用其他的构造方法，这样可以减少代码量，上述代码第⑤行 this(name,age,null)使用 this 调用其他构造方法。类似调用还有代码第⑥行的 this(name,30,d)和第⑦行的 this(name,null)。

注意 使用 this 调用其他构造方法时，this 语句一定是该构造方法的第一条语句。例如，在代码第⑦行之前调用 toString()方法则会发生错误。

11.5 销毁对象

对象不再使用时应该销毁。C++语言对象是通过 delete 语句手动释放,Java 语言对象是由垃圾回收器(garbage collection)收集然后释放,程序员不用关心释放的细节。自动内存管理是现代计算机语言发展趋势,如 C# 语言的垃圾回收、Objective-C 和 Swift 语言的 ARC(内存自动引用计数管理)。

垃圾回收器的工作原理是:当一个对象的引用不存在时,认为该对象不再需要,垃圾回收器自动扫描对象的动态内存区,把没有引用的对象作为垃圾收集起来并释放。

本章小结

通过对本章的学习,可以了解如何创建 Java 对象,理解构造方法的作用。此外,还介绍了 this 关键字的使用,以及如何销毁对象。

11.6 同步练习

1. 下面是有关子类继承父类构造方法的描述,其中正确的是(　　)。
   A. 创建子类的对象时,先调用子类自己的构造方法,然后调用父类的构造方法
   B. 子类无条件地继承父类不含参数的构造方法
   C. 子类必须通过 super 关键字调用父类的构造方法
   D. 子类无法继承父类的构造方法
2. 判断对错。
(1) 关键字 this 代表当前对象。(　　)
(2) 构造方法能继承,也能被重载。(　　)

第 12 章 继承与多态

CHAPTER 12

类的继承性是面向对象语言的基本特性,多态性的前提是继承性。Java 支持继承性和多态性。本章讨论 Java 继承性和多态性。

12.1 Java 中的继承

为了了解继承性,先看这样一个场景:一位面向对象的程序员小赵,在编程过程中需要描述和处理个人信息,于是定义了类 Person,代码如下所示:

```
//Person.java 文件
package com.a51work6;

import java.util.Date;

public class Person {

    //名字
    private String name;
    //年龄
    private int age;
    //出生日期
    private Date birthDate;

    public String getInfo() {
        return "Person [name=" + name
                + ", age=" + age
                + ", birthDate=" + birthDate + "]";
    }

}
```

一周以后,小赵又遇到了新的需求,需要描述和处理学生信息,于是他又定义了一个新的类 Student,代码如下所示:

```
//Student.java 文件
package com.a51work6;
```

```
import java.util.Date;

public class Student {

    //所在学校
    public String school;
    //名字
    private String name;
    //年龄
    private int age;
    //出生日期
    private Date birthDate;

    public String getInfo() {
        return "Person [name = " + name
                + ", age = " + age
                + ", birthDate = " + birthDate + "]";
    }
}
```

很多人会认为小赵的做法能够理解并相信这是可行的，但问题在于 Student 和 Person 两个类的结构太接近了，后者只比前者多了一个属性 school，却要重复定义其他所有的内容，实在让人“不甘心”。Java 提供了解决类似问题的机制，那就是类的继承，代码如下所示：

```
//Student.java 文件
package com.a51work6;

import java.util.Date;

public class Student extends Person {
    //所在学校
    private String school;
}
```

Student 类继承了 Person 类中的所有成员变量和方法，从上述代码可见继承使用的关键字是 extends，extends 后面的 Person 是父类。

如果在类的声明中没有使用 extends 关键字指明其父类，则默认父类为 Object 类，java.lang.Object 类是 Java 的根类，所有 Java 类包括数组都直接或间接继承了 Object 类，在 Object 类中定义了一些有关面向对象机制的基本方法，如 equals()、toString() 和 finalize() 等。

提示 一般情况下，一个子类只能继承一个父类，这称为“单继承”。但有的情况下一个子类可以有多个不同的父类，这称为“多重继承”。在 Java 中，类的继承只能是单继承，而多重继承可以通过实现多个接口实现。也就是说，在 Java 中，一个类只能继承一个父类，但是可以实现多个接口。

提示 面向对象分析与设计(OOAD)时,会用到 UML 图[①],其中类图非常重要,用来描述系统静态结构。Student 继承 Person 的类图如图 12-1 所示。类图中的各个元素说明如图 12-2 所示,类用矩形表示,一般分为上、中、下 3 个部分,上部分是类名,中部分是成员变量,下部分是成员方法。实线+空心箭头表示继承关系,箭头指向父类,箭头末端是子类。UML 类图中还有很多关系,如图 12-3 所示,如虚线+空心箭头表示实现关系,箭头指向接口,箭头末端是实现类。

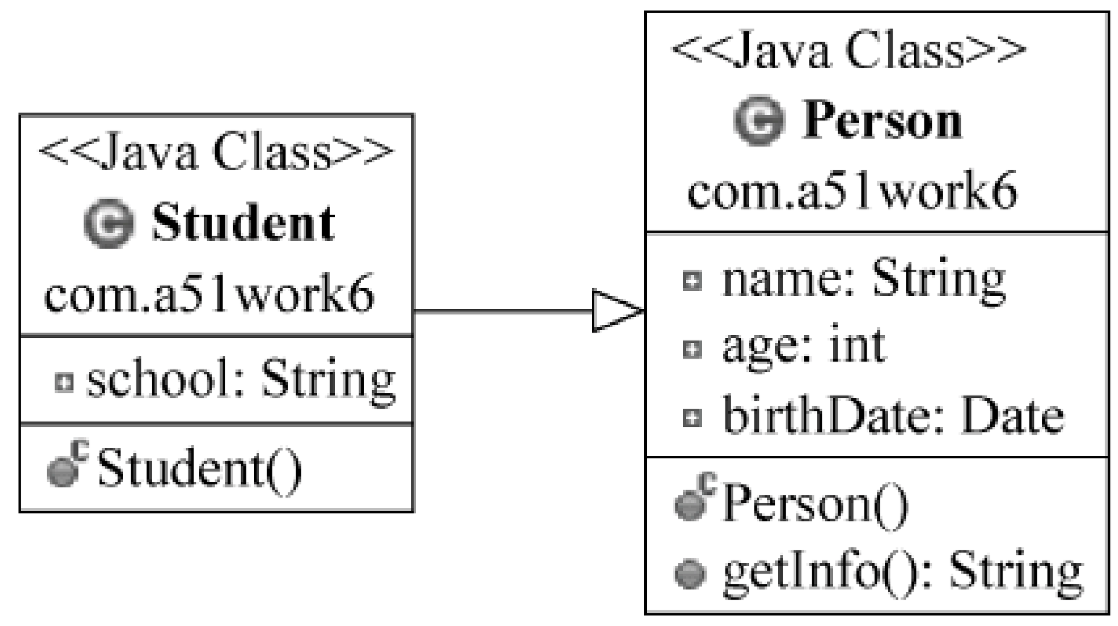


图 12-1 Student 继承 Person 的类图

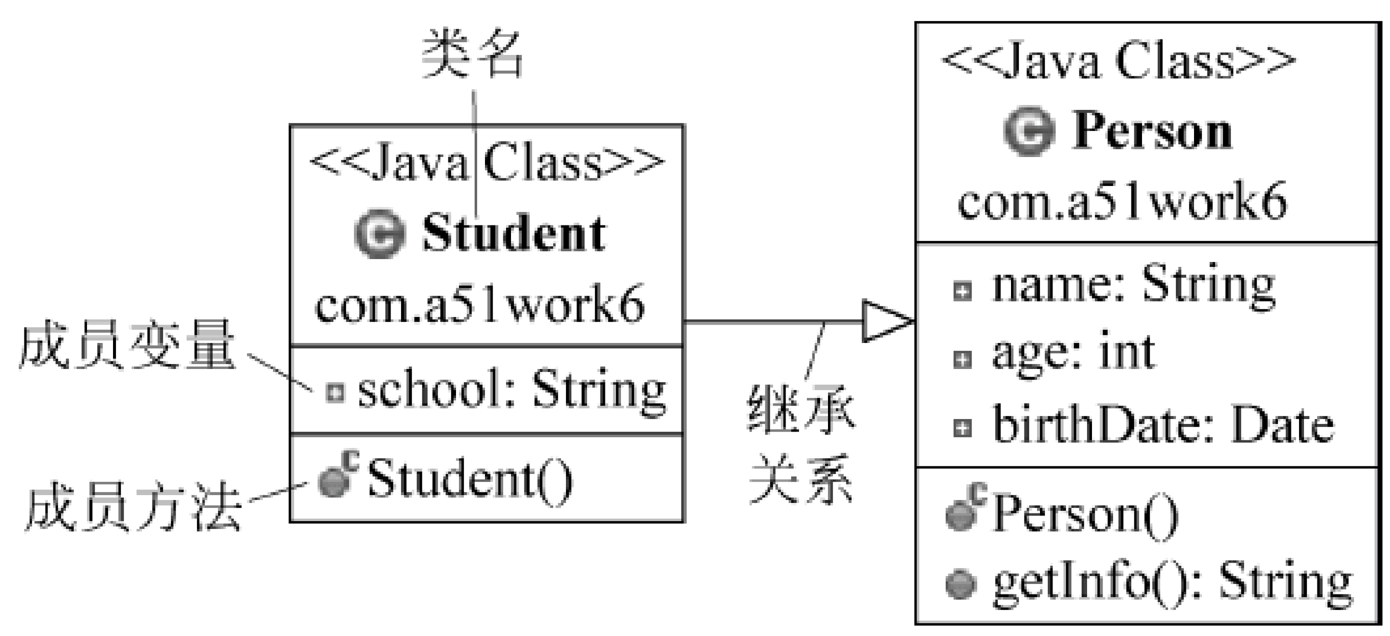


图 12-2 类图中元素

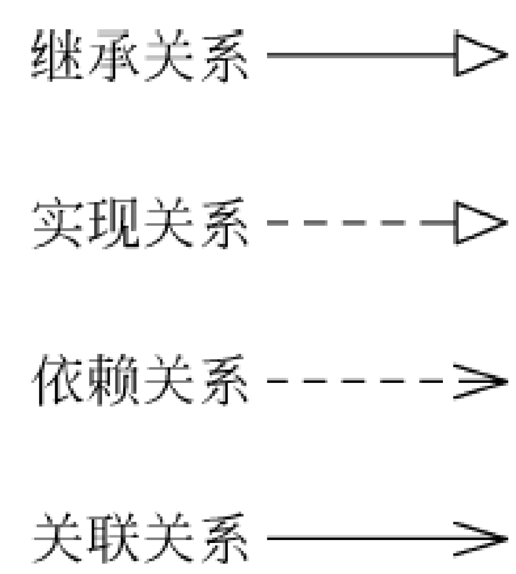


图 12-3 元素之间关系

12.2 调用父类构造方法

当子类实例化时,不仅需要初始化子类成员变量,也需要初始化父类成员变量,初始化父类成员变量需要调用父类构造方法,子类使用 super 关键字调用父类构造方法。

下面看一个示例。现有父类 Person 和子类 Student,它们的类图如图 12-4 所示。

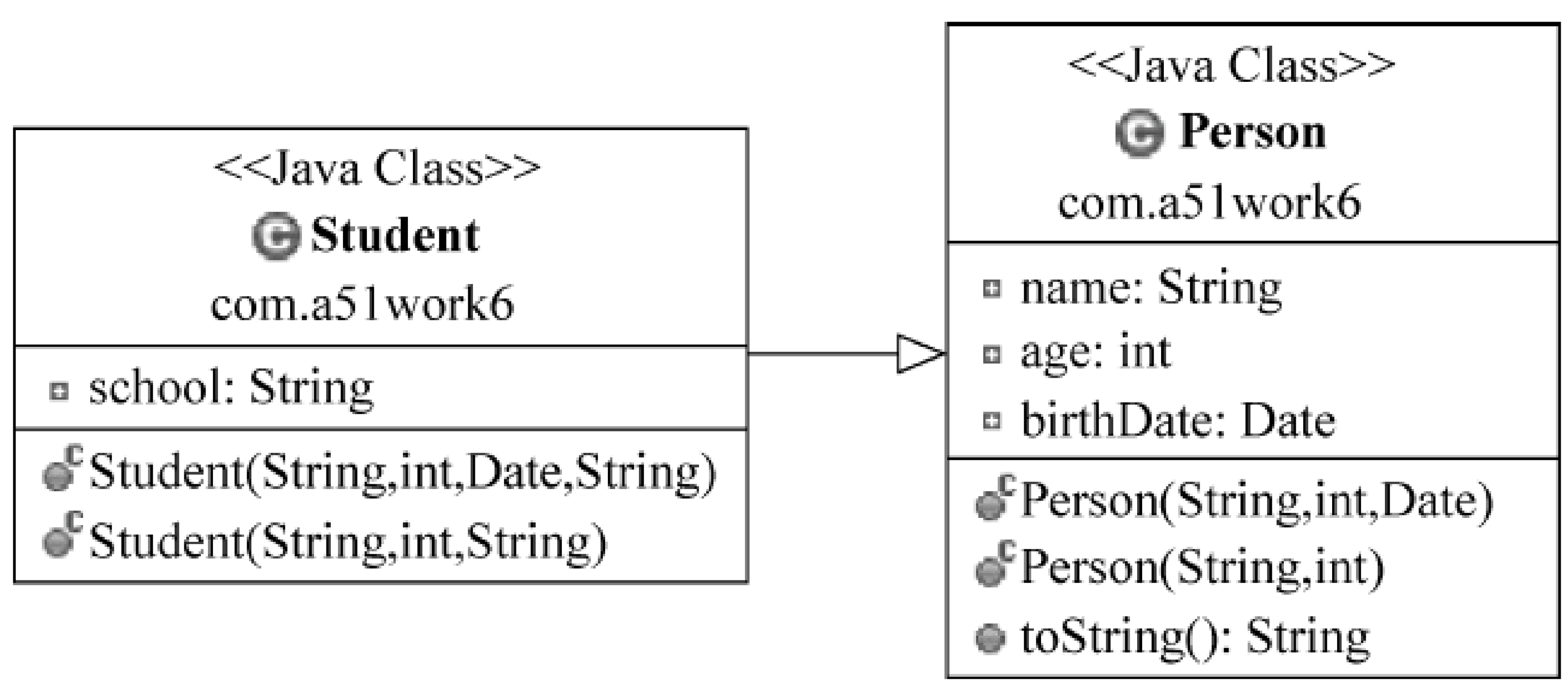


图 12-4 Person 和 Student 类图

父类 Person 代码如下:

```
//Person.java 文件
package com.a51work6;
```

① UML 是 Unified Modeling Language 的缩写,即统一标准建模语言。它是集成了各种优秀的建模方法发展而来的。UML 常用的有例图、协作图、活动图、序列图、部署图、构件图、类图、状态图等。

```
import java.util.Date;

public class Person {

    //名字
    private String name;
    //年龄
    private int age;
    //出生日期
    private Date birthDate;

    //3 个参数构造方法
    public Person(String name, int age, Date d) {
        this.name = name;
        this.age = age;
        birthDate = d;
    }

    public Person(String name, int age) {
        //调用 3 个参数构造方法
        this(name, age, new Date());
    }
    …
}
```

子类 Student 代码如下：

```
//Student.java 文件
package com.a51work6;

import java.util.Date;

public class Student extends Person {

    //所在学校
    private String school;

    public Student(String name, int age, Date d, String school) {
        super(name, age, d);                                          ①
        this.school = school;
    }

    public Student(String name, int age, String school) {
        //this.school = school;                       //编译错误
        super(name, age);                                             ②
        this.school = school;
    }

    public Student(String name, String school) {      //编译错误      ③
        //super(name, 30);
        this.school = school;
    }
}
```

在 Student 子类代码第①行和第②行是调用父类构造方法，代码第①行 super(name, age, d)语句是调用父类的 Person(String name, int age, Date d)构造方法，代码第②行 super(name, age)语句是调用父类的 Person(String name, int age)构造方法。

提示 super 语句必须位于子类构造方法的第一行。

代码第③行构造方法由于没有 super 语句，编译器会试图调用父类的默认构造方法(无参数构造方法)，但是父类 Person 并没有默认构造方法，因此会发生编译错误。解决这个编译错误有如下 3 种办法。

(1) 在父类 Person 中添加默认构造方法，子类 Student 会隐式调用父类的默认构造方法。

(2) 在子类 Student 构造方法中添加 super 语句，显式调用父类构造方法，super 语句必须是第一条语句。

(3) 在子类 Student 构造方法中添加 this 语句，显式调用当前对象的其他构造方法，this 语句必须是第一条语句。

12.3 成员变量隐藏和方法覆盖

子类继承父类后，在子类中有可能声明了与父类一样的成员变量或方法，那么会出现什么情况呢？

12.3.1 成员变量隐藏

子类成员变量与父类一样，会屏蔽父类中的成员变量，称为成员变量隐藏。示例代码如下：

```
//ParentClass.java 文件
package com.a51work6;

class ParentClass {
    //x 成员变量
    int x = 10;                                        ①
}

class SubClass extends ParentClass {
    //屏蔽父类 x 成员变量
    int x = 20;                                        ②

    public void print() {
        //访问子类对象 x 成员变量
        System.out.println("x = " + x);                ③
        //访问父类 x 成员变量
        System.out.println("super.x = " + super.x);    ④
    }
}
```

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[ ] args) {
        //实例化子类 SubClass
        SubClass pObj = new SubClass();
        //调用子类 print 方法
        pObj.print();
    }
}
```

运行结果如下：

```
x = 20
super.x = 10
```

上述代码第①行是在 ParentClass 类声明 x 成员变量，在它的子类 SubClass 代码第②行也声明了 x 成员变量，它会屏蔽父类中的 x 成员变量，代码第③行的 x 是子类中的 x 成员变量。如果要调用父类中的 x 成员变量，则需要 super 关键字，见代码第④行的 super. x。

12.3.2 方法覆盖

如果子类方法完全与父类方法相同，即相同的方法名、相同的参数列表和相同的返回值，只是方法体不同，这称为子类覆盖(override)父类方法。

示例代码如下：

```
//ParentClass.java 文件
package com.a51work6;

class ParentClass {
    //x 成员变量
    int x;

    protected void setValue() {                    ①
        x = 10;
    }
}

class SubClass extends ParentClass {
    //屏蔽父类 x 成员变量
    int x;

    @Override
    public void setValue() {   //覆盖父类方法      ②
        //访问子类对象 x 成员变量
```

```
            x = 20;
            //调用父类 setValue()方法
            super.setValue();
        }

        public void print() {
            //访问子类对象 x 成员变量
            System.out.println("x = " + x);
            //访问父类 x 成员变量
            System.out.println("super.x = " + super.x);
        }
}
```

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        //实例化子类 SubClass
        SubClass pObj = new SubClass();
        //调用 setValue 方法
        pObj.setValue();
        //调用子类 print 方法
        pObj.print();
    }
}
```

运行结果如下：

```
x = 20
super.x = 10
```

上述代码第①行是在 ParentClass 类中声明 setValue 方法，在它的子类 SubClass 代码第②行覆盖父类中的 setValue 方法。在声明方法时添加@Override 注解，@Override 注解不是方法覆盖必需的，它只是锦上添花，但添加@Override 注解有以下两个好处。

（1）提高程序的可读性。

（2）编译器检查@Override 注解的方法在父类中是否存在，如果不存在，则报错。

注意　方法覆盖时应遵循的原则如下。

（1）覆盖后的方法不能比原方法有更严格的访问控制（可以相同）。例如，将代码第②行访问控制 public 修改为 private，那么会发生编译错误，因为父类原方法是 protected。

（2）覆盖后的方法不能比原方法产生更多的异常。

12.4 多态

在面向对象程序设计中，多态是一个非常重要的特性，理解多态有利于进行面向对象的分析与设计。

12.4.1 多态概念

发生多态要有以下 3 个前提条件。

(1) 继承。多态发生在子类和父类之间。

(2) 覆盖。子类覆盖了父类的方法。

(3) 声明的变量类型是父类类型，但实例则指向子类实例。

下面通过一个示例理解什么是多态。如图 12-5 所示，父类 Figure(几何图形)类有一个 onDraw(绘图)方法，Figure(几何图形)有两个子类 Ellipse(椭圆形)和 Triangle(三角形)，Ellipse 和 Triangle 覆盖 onDraw 方法。Ellipse 和 Triangle 都有 onDraw 方法，但具体实现的方式不同。

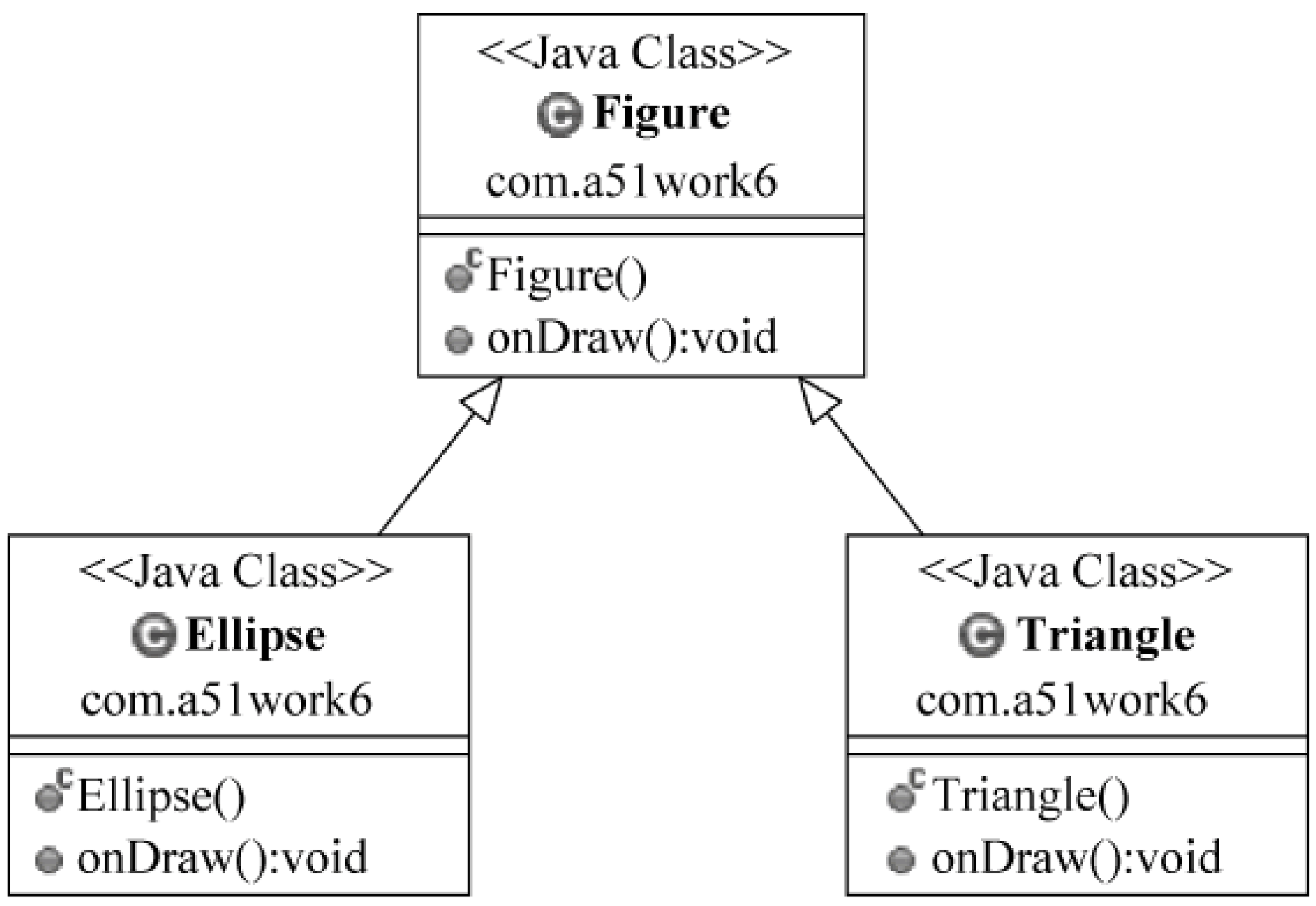


图 12-5 几何图形类图

具体代码如下：

```
//Figure.java 文件
package com.a51work6;

public class Figure {

    //绘制几何图形方法
    public void onDraw() {
        System.out.println("绘制 Figure…");
    }
}

//Ellipse.java 文件
package com.a51work6;
```

```
//几何图形:椭圆形
public class Ellipse extends Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制椭圆形…");
    }

}

//Triangle.java 文件
package com.a51work6;

//几何图形:三角形
public class Triangle extends Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制三角形…");
    }
}
```

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;
public class HelloWorld {
    public static void main(String[] args) {

        //f1 变量是父类类型,指向父类实例
        Figure f1 = new Figure();                ①
        f1.onDraw();

        //f2 变量是父类类型,指向子类实例,发生多态
        Figure f2 = new Triangle();              ②
        f2.onDraw();

        //f3 变量是父类类型,指向子类实例,发生多态
        Figure f3 = new Ellipse();               ③
        f3.onDraw();

        //f4 变量是子类类型,指向子类实例
        Triangle f4 = new Triangle();            ④
        f4.onDraw();

    }
}
```

上述代码第②行和第③行符合多态的 3 个前提，因此会发生多态。而代码第①行和第④行都不符合，没有发生多态。

运行结果如下：

```
绘制 Figure…
绘制三角形…
绘制椭圆形…
绘制三角形…
```

从运行结果可知，多态发生时，Java 虚拟机运行时根据引用变量指向的实例调用它的方法，而不是根据引用变量的类型调用。

12.4.2 引用类型检查

有时候需要在运行时判断一个对象是否属于某个引用类型，这时可以使用 instanceof 运算符。instanceof 运算符语法格式如下：

```
obj instanceof type
```

其中，obj 是一个对象，type 是引用类型，如果 obj 对象是 type 引用类型实例，则返回 true，否则返回 false。

为了介绍引用类型检查，先看一个示例。图 12-6 所示的类图展示了继承层次树，Person 类是根类，Student 是 Person 的直接子类，Worker 是 Person 的直接子类。

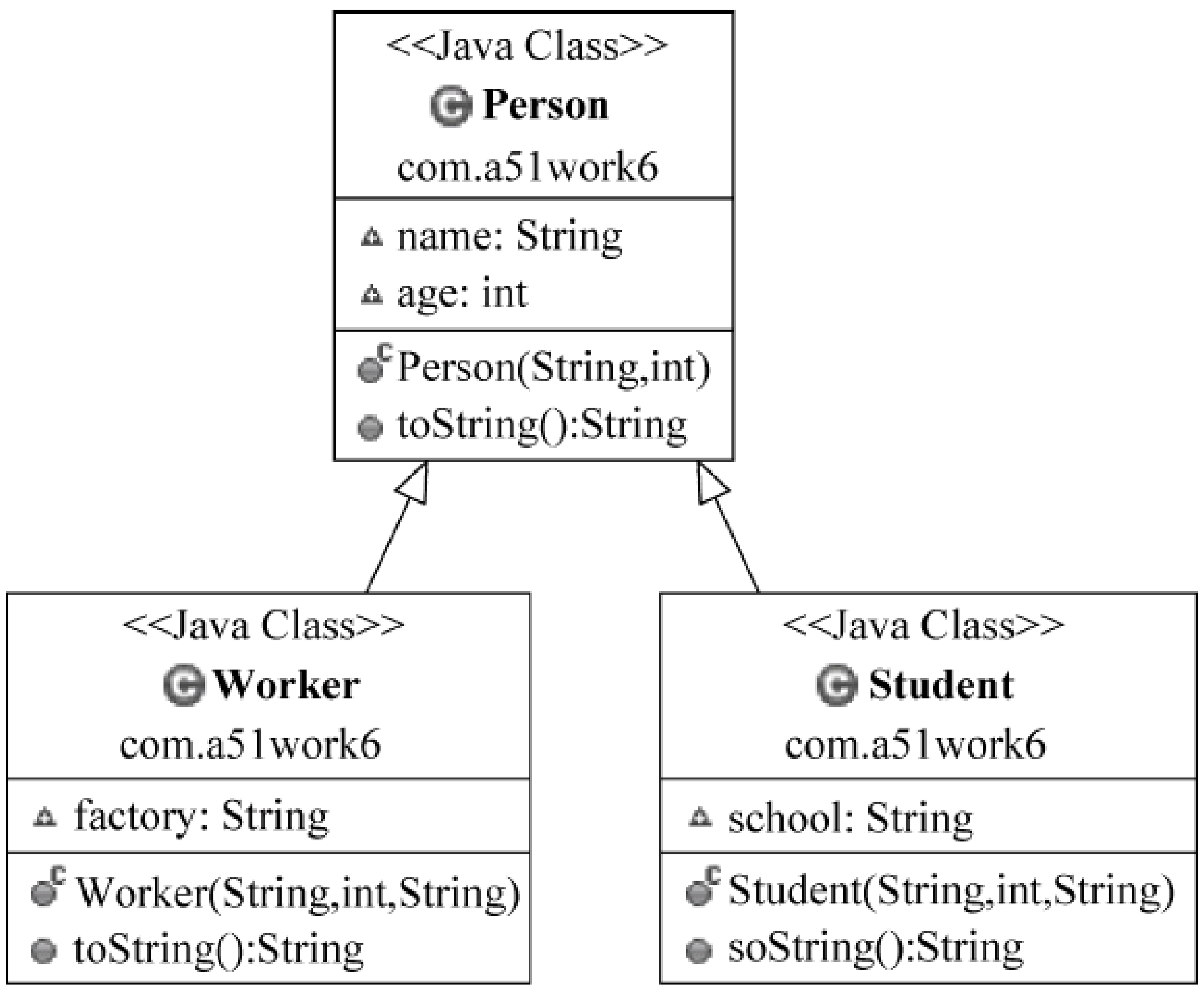


图 12-6　继承关系类图

继承层次树中具体实现代码如下：

```
//Person.java 文件
package com.a51work6;
public class Person {

    String name;
```

```
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name
                + ", age=" + age + "]";
    }
}

//Worker.java 文件
package com.a51work6;
public class Worker extends Person {

    String factory;

    public Worker(String name, int age, String factory) {
        super(name, age);
        this.factory = factory;
    }

    @Override
    public String toString() {
        return "Worker [factory=" + factory
                + ", name=" + name
                + ", age=" + age + "]";
    }
}

//Student.java 文件
package com.a51work6;
public class Student extends Person {

    String school;

    public Student(String name, int age, String school) {
        super(name, age);
        this.school = school;
    }

    @Override
    public String toString() {
        return "Student [school=" + school
                + ", name=" + name
                + ", age=" + age + "]";
    }

}
```

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Student student1 = new Student("Tom", 18, "清华大学");                    ①
        Student student2 = new Student("Ben", 28, "北京大学");
        Student student3 = new Student("Tony", 38, "香港大学");                   ②

        Worker worker1 = new Worker("Tom", 18, "钢厂");                           ③
        Worker worker2 = new Worker("Ben", 20, "电厂");                           ④

        Person[] people = { student1, student2, student3, worker1, worker2 };    ⑤

        int studentCount = 0;
        int workerCount = 0;

        for (Person item : people) {                                             ⑥
            if (item instanceof Worker) {                                        ⑦
                workerCount++;
            } else if (item instanceof Student) {                                ⑧
                studentCount++;
            }
        }
        System.out.printf("工人人数:%d,学生人数:%d", workerCount, studentCount);
    }
}
```

上述代码第①行和第②行创建了 3 个 Student 实例，代码第③行和第④行创建了 2 个 Worker 实例，然后程序把这 5 个实例放入 people 数组中。

代码第⑥行使用增强 for 循环遍历 people 数组集合，当从 people 数组中取出元素时，元素类型是 People 类型，但是实例不知道是哪个子类(Student 和 Worker)实例。代码第⑦行的 item instanceof Worker 表达式判断数组中的元素是否是 Worker 实例；类似地，第⑧行的 item instanceof Student 表达式判断数组中的元素是否是 Student 实例。

输出结果如下：

```
工人人数:2,学生人数:3
```

12.4.3 引用类型转换

在 5.7 节介绍过数值类型相互转换，引用类型可以进行转换，但并不是所有的引用类型都能互相转换，只有属于同一棵继承层次树中的引用类型才可以转换。

在上一节示例上修改 HelloWorld.java 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[ ] args) {

        Person p1 = new Student("Tom", 18, "清华大学");
        Person p2 = new Worker("Tom", 18, "钢厂");

        Person p3 = new Person("Tom", 28);
        Student p4 = new Student("Ben", 40, "清华大学");
        Worker p5 = new Worker("Tony", 28, "钢厂");
        …
    }
}
```

上述代码创建了 5 个实例 p1、p2、p3、p4 和 p5，它们的类型都是 Person 继承层次树中的引用类型，p1 和 p4 是 Student 实例，p2 和 p5 是 Worker 实例，p3 是 Person 实例。首先，对象类型转换一定发生在继承的前提下，p1 和 p2 都声明为 Person 类型，而实例是由 Person 子类型实例化的。

表 12-1 归纳了 p1、p2、p3、p4 和 p5 这 5 个实例与 Person 和 Worker、Student 这 3 种类型之间的转换关系。

表 12-1　类型转换

对　象	Person 类型	Worker 类型	Student 类型	说　明
p1	支持	不支持	支持(向下转型)	类型：Person 实例：Student
p2	支持	支持(向下转型)	不支持	类型：Person 实例：Worker
p3	支持	不支持	不支持	类型：Person 实例：Person
p4	支持(向上转型)	不支持	支持	类型：Student 实例：Student
p5	支持(向上转型)	支持	不支持	类型：Worker 实例：Worker

作为这段程序的编写者知道 p1 本质上是 Student 实例，但是表面上看是 Person 类型，编译器也无法推断 p1 的实例是 Person、Student 还是 Worker。此时可以使用 instanceof 操作符来判断它是哪一类的实例。

引用类型转换也是通过小括号运算符实现，类型转换有两个方向：将父类引用类型变量转换为子类类型，这种转换称为向下转型(downcast)；将子类引用类型变量转换为父类类型，这种转换称为向上转型(upcast)。向下转型需要强制转换，而向上转型是自动的。

下面通过示例详细说明一下向下转型和向上转型，在 HelloWorld.java 的 main 方法中添加如下代码：

```
//向上转型
Person p = (Person) p4;                                          ①

//向下转型
Student p11 = (Student) p1;                                      ②
Worker p12 = (Worker) p2;                                        ③

//Student p111 = (Student) p2;          //运行时异常              ④
if (p2 instanceof Student) {
    Student p111 = (Student) p2;
}
//Worker p121 = (Worker) p1;            //运行时异常              ⑤
if (p1 instanceof Worker) {
    Worker p121 = (Worker) p1;
}
//Student p131 = (Student) p3;          //运行时异常              ⑥
if (p3 instanceof Student) {
    Student p131 = (Student) p3;
}
```

上述代码第①行将 p4 对象转换为 Person 类型，p4 本质上是 Student 实例，这是向上转型，这种转换是自动的，其实不需要小括号(Person)进行强制类型转换。

代码第②行和第③行是向下类型转换，它们的转型都能成功。而代码第④、⑤、⑥行都会发生运行时异常 ClassCastException，如果不能确定实例是哪一种类型，可以在转型之前使用 instanceof 运算符判断一下。

12.5 再谈 final 关键字

在前面的学习过程中，为了声明常量使用过 final 关键字，在 Java 中 final 关键字的作用还有很多，final 关键字能修饰变量、方法和类。下面详细加以说明。

12.5.1 final 修饰变量

final 修饰的变量即成为常量，只能赋值一次，但是 final 所修饰的局部变量和成员变量有所不同。

(1) final 修饰的局部变量必须使用之前被赋值一次才能使用。

(2) final 修饰的成员变量在声明时没有赋值的称为"空白 final 变量"。空白 final 变量必须在构造方法或静态代码块中初始化。

final 修饰变量示例代码如下：

```
//FinalDemo.java 文件
package com.a51work6;

class FinalDemo {

    void doSomething() {
```

```
            //没有在声明的同时赋值
            final int e;                                      ①
            //只能赋值一次
            e = 100;                                          ②
            System.out.print(e);
            //声明的同时赋值
            final int f = 200;                                ③
        }

        //实例常量
        final int a = 5;                //直接赋值            ④
        final int b;                    //空白 final 变量     ⑤

        //静态常量
        final static int c = 12;        //直接赋值            ⑥
        final static int d;             //空白 final 变量     ⑦

        //静态代码块
        static {
            //初始化静态变量
            d = 32;                                           ⑧
        }

        //构造方法
        FinalDemo() {
            //初始化实例变量
            b = 3;                                            ⑨
            //第二次赋值,会发生编译错误
            //b = 4;                                          ⑩
        }
    }
```

上述代码第①行和第③行是声明局部常量，其中第①行只是声明没有赋值，但必须在使用之前赋值(见代码第②行)，其实局部常量最好在声明的同时初始化。

代码第④、⑤、⑥和⑦行都声明成员常量。代码第④行和第⑤行是实例常量，如果是空白 final 变量(见代码第⑤行)，则需要在构造方法中初始化(见代码第⑨行)；代码第⑥行和第⑦行是静态常量，如果是空白 final 变量(见代码第⑦行)，则需要在静态代码块中初始化(见代码第⑧行)。

另外，无论哪种常量只能赋值一次。代码第⑩行为 b 常量赋值，因为之前 b 已经赋值过一次，因此这里会发生编译错误。

12.5.2　final 修饰类

final 修饰的类不能被继承。有时出于设计安全的目的，不想让自己编写的类被别人继

承，这时可以使用 final 关键字修饰父类。

示例代码如下：

```
//SuperClass.java 文件
package com.a51work6;

final class SuperClass {
}

class SubClass extends SuperClass {              //编译错误
}
```

在声明 SubClass 类时会发生编译错误。

12.5.3 final 修饰方法

final 修饰的方法不能被子类覆盖。有时也是出于设计安全的目的，父类中的方法不想被别人覆盖，这时可以使用 final 关键字修饰父类中的方法。

示例代码如下：

```
//SuperClass.java 文件
package com.a51work6;

class SuperClass {
    final void doSomething() {
        System.out.println("in SuperClass.doSomething()");
    }
}

class SubClass extends SuperClass {
    @Override
    void doSomething() {                     //编译错误
        System.out.println("in SubClass.doSomething()");
    }
}
```

子类中的 void doSomething()方法试图覆盖父类中的 void doSomething()方法，父类中的 void doSomething()方法是 final 修饰的方法，不能被子类覆盖的，因此会发生编译错误。

本章小结

本章首先介绍了 Java 中的继承概念，在继承时会发生方法的覆盖、变量的隐藏。然后介绍了 Java 中的多态概念，读者需要熟悉多态发生的条件，掌握引用类型检查和类型转换。最后还介绍了 final 关键字。

12.6 同步练习

1. 下列哪些说法是正确的？（   ）

   A. Java 语言只允许单一继承

   B. Java 语言只允许实现一个接口

   C. Java 语言不允许同时继承一个类并实现一个接口

   D. Java 语言的单一继承使得代码更加可靠

2. 现在有两个类：Person 与 Chinese，Chinese 试图继承 Person 类，如下项目中哪个是正确的写法？（   ）

   A. class Chinese extents Person{}　　B. class Chinese extants Person{}

   C. class Chinese extends Person{}　　D. class Chinese extands Person{}

3. 类 Teacher 和 Student 是类 Person 的子类，有如下代码：

```
Person p;
Teacher t;
Student s;
//假设 p、t、s 都是非空的
if(t instance of Person) { s = (Student)t; }
```

有关最后一条语句的说法正确的是（   ）。

   A. 将构造一个 Student 对象　　B. 表达式是合法的

   C. 表达式是错误的　　D. 编译时正确，但运行时错误

4. 有如下代码：

```
1) class Parent {
2)    private String name;
3)    public Parent(){}
4) }
5) public class Child extends Parent {
6)    private String department;
7)    public Child() {}
8)    public String getValue(){ return name; }
9)    public static void main(String arg[]) {
10)       Parent p = new Parent();
11)    }
12) }
```

下列哪些行会引起程序运行错误？（   ）

   A. 第 3 行　　B. 第 6 行　　C. 第 7 行　　D. 第 8 行

5. 判断对错：声明为 final 的方法不能在子类中重载。（   ）

6. 有如下代码：

```
public class Parent {
    int change() {}
}
class Child extends Parent { }
```

下列哪些方法可加入类 Child 中？（　　）

A. public int change(){}

B. int chang(int i){}

C. private int change(){}

D. abstract int chang(){}

第 13 章 抽象类与接口

CHAPTER 13

设计良好的软件系统应该具备可复用性和可扩展性，能够满足用户不断变更的需求。使用抽象类和接口是实现可复用性和可扩展性重要的设计手段。

13.1 抽象类

Java 语言提供了两种类：一种是具体类；另一种是抽象类。前面章节接触的类都是具体类。本节介绍抽象类。

13.1.1 抽象类的概念

在 12.4.1 节介绍多态时，使用过几何图形类示例，其中 Figure（几何图形）类中有一个 onDraw（绘图）方法，Figure 有两个子类 Ellipse（椭圆形）和 Triangle（三角形），Ellipse 和 Triangle 覆盖 onDraw 方法。

作为父类 Figure（几何图形）并不知道在实际使用时有多少个子类，目前有椭圆形和三角形，那么不同的用户需求可能会有矩形或圆形等其他几何图形，而 onDraw 方法只有确定是哪一个子类后才能具体实现。Figure 中的 onDraw 方法不能具体实现，所以只能是一个抽象方法。在 Java 中具有抽象方法的类称为抽象类，Figure 是抽象类，其中的 onDraw 方法是抽象方法。如图 13-1 所示类图中，Figure 是抽象类，Ellipse 和 Triangle 是 Figure 子类，实现 Figure 的抽象方法 onDraw。

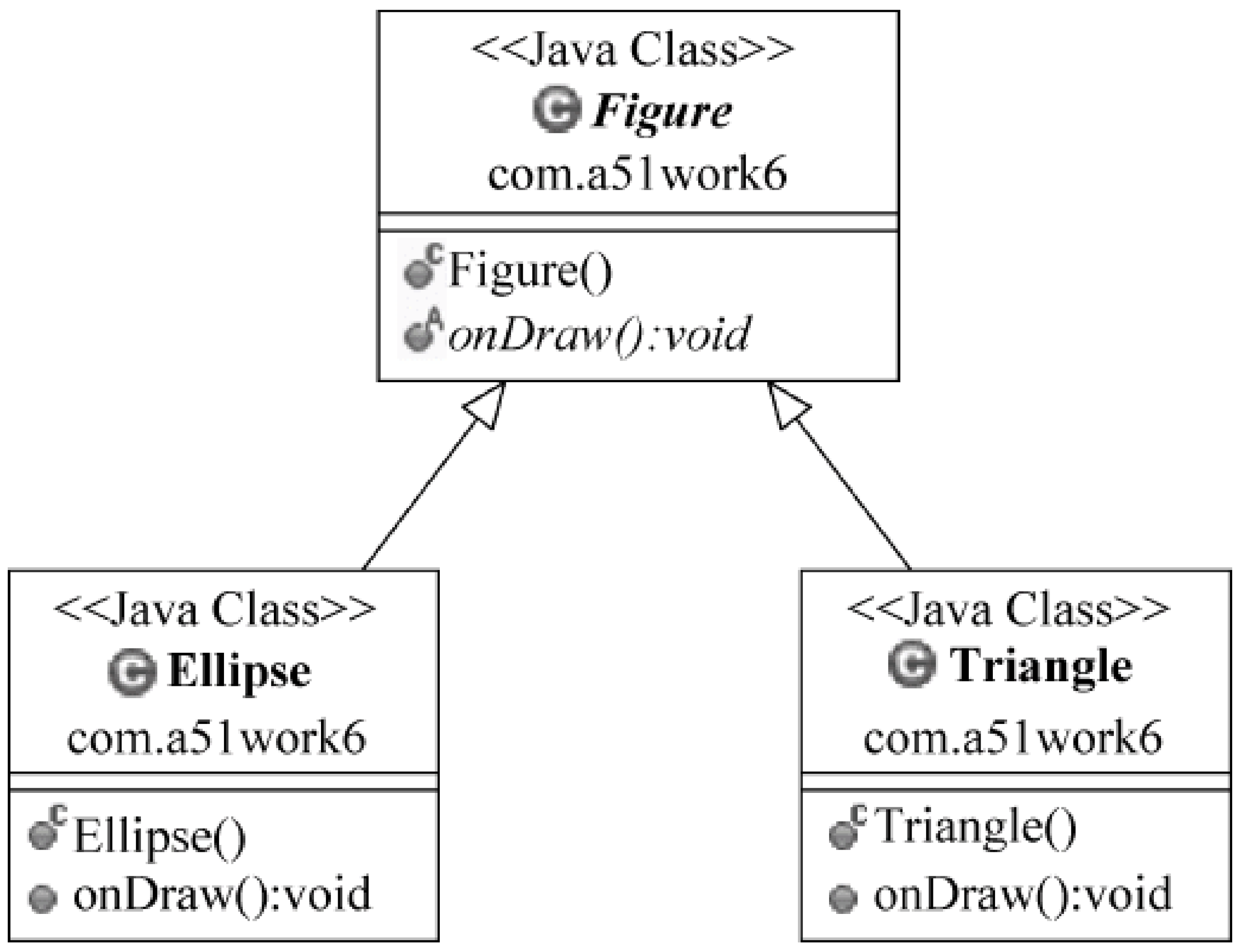


图 13-1 抽象类几何图形类图

提示 在UML类图中，抽象类和抽象方法的字体是斜体，如图13-1所示的Figure类和onDraw方法都是斜体。

13.1.2 抽象类声明和实现

在Java中抽象类和抽象方法的修饰符是abstract。声明抽象类Figure示例代码如下：

```
//Figure.java 文件
package com.a51work6;

public abstract class Figure {                    ①
    //绘制几何图形方法
    public abstract void onDraw();                ②
}
```

代码第①行是声明抽象类，在类前面加上abstract修饰符。代码第②行声明抽象方法，方法前面的修饰符也是abstract。注意抽象方法中只有方法的声明，没有方法的实现，即没有大括号({})部分。

注意 如果一个方法被声明为抽象的，那么这个类也必须声明为抽象的。而一个抽象类中可以有0～*n*个抽象方法，以及0～*n*个具体方法。

设计抽象方法的目的就是让子类来实现，否则抽象方法就没有任何意义。实现抽象类示例代码如下：

```
//Ellipse.java 文件
package com.a51work6;

//几何图形:椭圆形
public class Ellipse extends Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制椭圆形…");
    }
}

//Triangle.java 文件
package com.a51work6;

//几何图形:三角形
public class Triangle extends Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制三角形…");
    }
}
```

上述代码声明了两个具体类Ellipse和Triangle，它们实现(覆盖)了抽象类Figure的抽

象方法 onDraw。

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //f1 变量是父类类型,指向子类实例,发生多态
        Figure f1 = new Triangle();
        f1.onDraw();

        //f2 变量是父类类型,指向子类实例,发生多态
        Figure f2 = new Ellipse();
        f2.onDraw();
    }
}
```

上述代码中实例化两个具体类 Triangle 和 Ellipse，对象 f1 和 f2 是 Figure 引用类型。

注意　抽象类不能被实例化，只有具体类才能被实例化。

13.2　接口

比抽象类更加抽象的是接口，在接口中所有的方法都是抽象的。

提示　Java 8 之后接口中新增加了默认方法，因此“接口中所有的方法都是抽象的”这个提法在 Java 8 之后是有待商榷的。

13.2.1　接口概念

其实 13.1.1 节抽象类 Figure 可以更加彻底，即 Figure 接口，接口中所有方法都是抽象的，而且接口可以有成员变量。将 13.1.1 节几何图形类改成接口后，类图如图 13-2 所示。

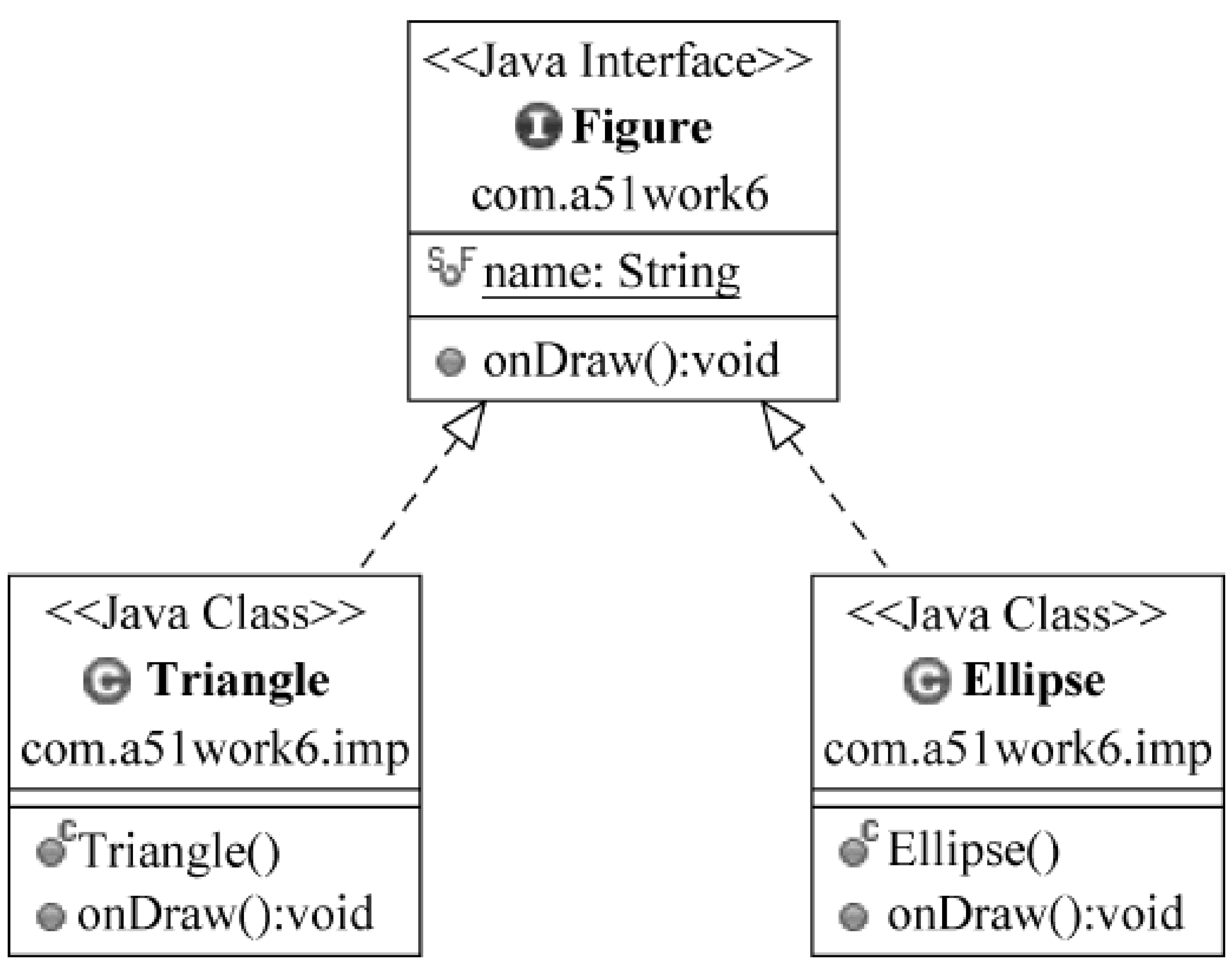


图 13-2　接口几何图形类图

提示 在 UML 类图中，接口的图标是“I”，如图 13-2 所示中的 Figure 接口；类的图标是“C”，如图 13-2 所示中的 Triangle 接口。

13.2.2 接口声明和实现

在 Java 中接口的声明使用的关键字是 interface。声明接口 Figure 示例代码如下：

```
//Figure.java 文件
package com.a51work6;

public interface Figure {                                    ①
    //接口中静态成员变量
    String name = "几何图形";     //省略 public static final     ②

    //绘制几何图形方法
    void onDraw();               //省略 public                  ③
}
```

代码第①行声明 Figure 接口，声明接口使用 interface 关键字，interface 前面的修饰符是 public 或省略。public 是公有访问级别，可以在任何地方访问。省略时是默认访问级别，只能在当前包中访问。

代码第②行声明接口中的成员变量，在接口中成员变量都是静态成员变量，即省略了 public static final 修饰符。代码第③行声明抽象方法，即省略了 public 关键字。

某个类实现接口时，要在声明时使用 implements 关键字，当实现多个接口时，其之间用逗号(,)分隔。实现接口时要实现接口中声明的所有方法。

实现接口 Figure 示例代码如下：

```
//Ellipse.java 文件
package com.a51work6.imp;

import com.a51work6.Figure;

//几何图形:椭圆形
public class Ellipse implements Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制椭圆形…");
    }
}

//Triangle.java 文件
package com.a51work6.imp;

import com.a51work6.Figure;

//几何图形:三角形
```

```
public class Triangle implements Figure {

    //绘制几何图形方法
    @Override
    public void onDraw() {
        System.out.println("绘制三角形…");
    }
}
```

上述代码声明了两个具体类 Ellipse 和 Triangle,它们实现了接口 Figure 中的抽象方法 onDraw。

调用代码如下:

```
//HelloWorld.java 文件
import com.a51work6.imp.Ellipse;
import com.a51work6.imp.Triangle;

public class HelloWorld {

    public static void main(String[] args) {

        //f1 变量是父类类型,指向子类实例,发生多态
        Figure f1 = new Triangle();
        f1.onDraw();
        System.out.println(f1.name);                ①
        System.out.println(Figure.name);            ②

        //f2 变量是父类类型,指向子类实例,发生多态
        Figure f2 = new Ellipse();
        f2.onDraw();
    }
}
```

上述代码中实例化两个具体类 Triangle 和 Ellipse,对象 f1 和 f2 是 Figure 接口引用类型。接口 Figure 中声明了成员变量,它是静态成员变量,代码第①行和第②行是访问 name 静态变量。

注意　接口与抽象类一样都不能被实例化。

13.2.3　接口与多继承

在 C++语言中一个类可以继承多个父类,但这会有潜在的风险,如果两个父类有相同的方法,那么子类将继承哪一个父类方法呢? 这就是 C++多继承所导致的冲突问题。

在 Java 中只允许继承一个类,但可实现多个接口。通过实现多个接口方式满足多继承的设计需求。如果多个接口中即便有相同方法,它们也都是抽象的,子类实现它们不会有冲突。

图 13-3 所示是多继承类图,其中有两个接口 InterfaceA 和 InterfaceB。从类图中可以看到,两个接口中都有一个相同的方法 void methodB()。AB 实现了这两个接口,继承了

Object 父类。

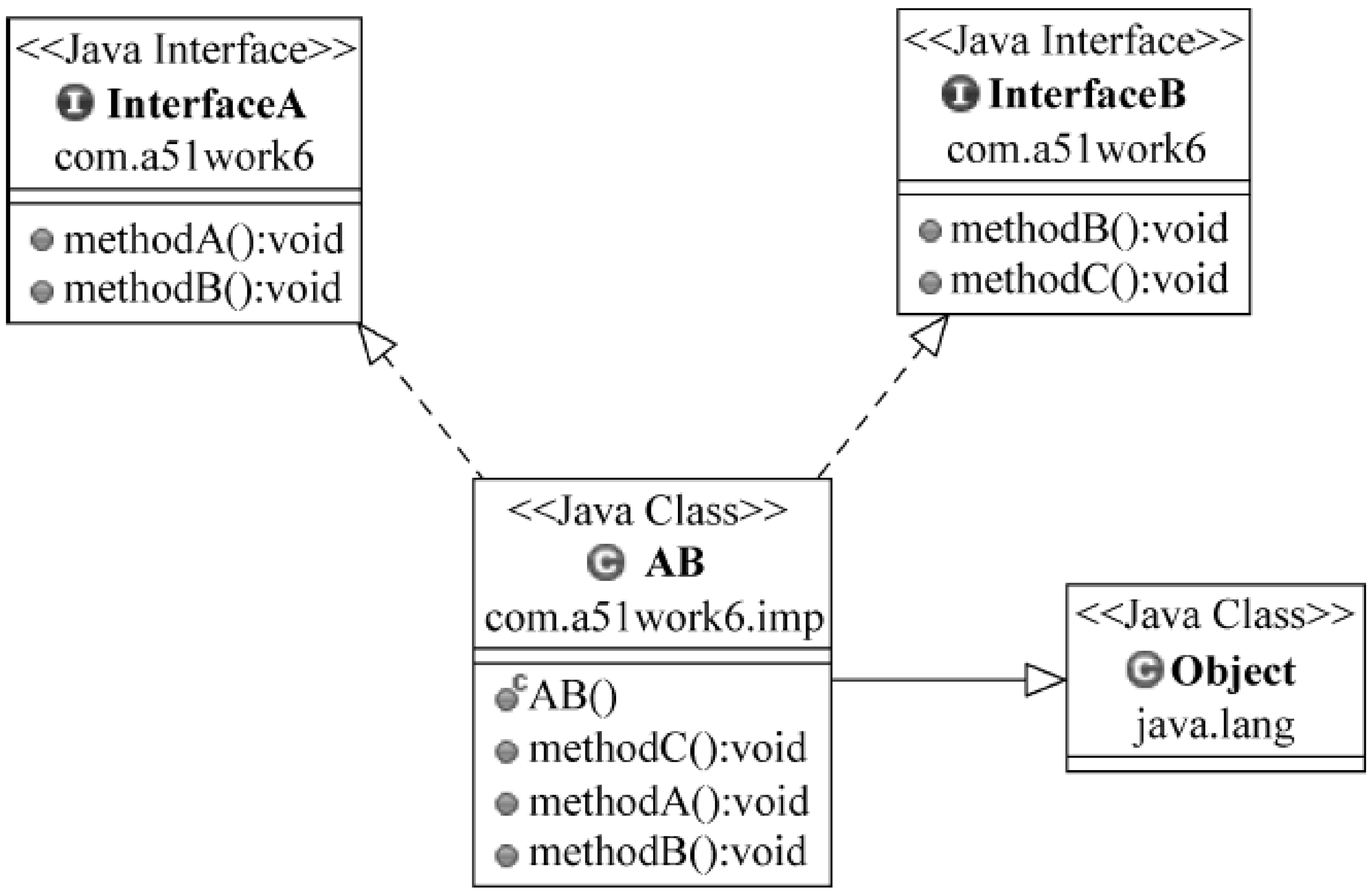


图 13-3 多继承类图

接口 InterfaceA 和 InterfaceB 代码如下：

```
//InterfaceA.java 文件
package com.a51work6;

public interface InterfaceA {

    void methodA();

    void methodB();
}

//InterfaceB.java 文件
package com.a51work6;

public interface InterfaceB {

    void methodB();

    void methodC();
}
```

从代码中可见两个接口都有两个方法，其中方法 methodB()完全相同。

实现接口 InterfaceA 和 InterfaceB 的 AB 类代码如下：

```
//AB.java 文件
package com.a51work6.imp;

import com.a51work6.InterfaceA;
import com.a51work6.InterfaceB;

public class AB extends Object implements InterfaceA, InterfaceB {        ①
```

```
        @Override
        public void methodC() {
        }

        @Override
        public void methodA() {
        }

        @Override
        public void methodB() {                    ②
        }
}
```

在 AB 类中的代码第②行实现 methodB()方法。注意在 AB 类声明时实现两个接口，接口之间使用逗号(,)分隔，见代码第①行。

13.2.4　接口继承

Java 语言中允许接口和接口之间继承。由于接口中的方法都是抽象方法，所以继承之后也不需要做什么，因此接口之间的继承要比类之间的继承简单得多。如图 4-4 所示，其中 InterfaceB 继承了 InterfaceA，在 InterfaceB 中还覆盖了 InterfaceA 中的 methodB()方法。ABC 是 InterfaceB 接口的实现类，从图 13-4 中可见，ABC 需要实现 InterfaceA 和 InterfaceB 接口中的所有方法。

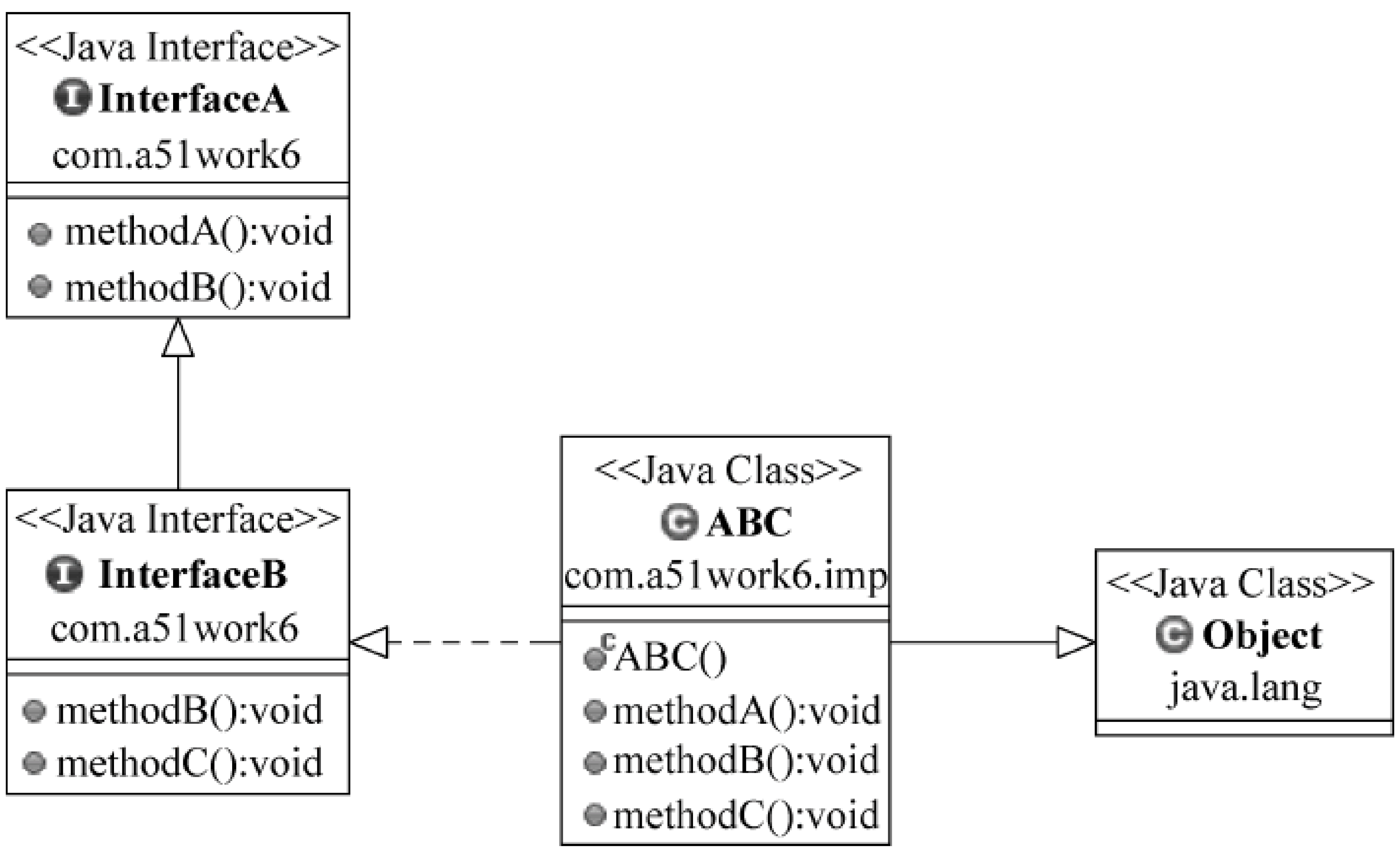


图 13-4　接口继承类图

接口 InterfaceA 和 InterfaceB 代码如下：

```
//InterfaceA.java 文件
package com.a51work6;

public interface InterfaceA {

    void methodA();
```

```
    void methodB();
}

//InterfaceB.java 文件
package com.a51work6;

public interface InterfaceB extends InterfaceA {

    @Override
    void methodB();

    void methodC();
}
```

InterfaceB 继承了 InterfaceA,声明时也使用 extends 关键字。InterfaceB 中的 methodB()覆盖了 InterfaceA,事实上在接口中覆盖方法并没有实际意义,因为它们都是抽象的,都是留给子类实现的。

实现接口 InterfaceB 的 ABC 类代码如下:

```
//ABC.java 文件
package com.a51work6.imp;

import com.a51work6.InterfaceB;

public class ABC implements InterfaceB {

    @Override
    public void methodA() {
    }

    @Override
    public void methodB() {
    }

    @Override
    public void methodC() {
    }
}
```

ABC 类实现了接口 InterfaceB,事实上是实现 InterfaceA 和 InterfaceB 中所有方法,相当于同时实现 InterfaceA 和 InterfaceB 接口。

13.2.5 Java 8 新特性默认方法和静态方法

在 Java 8 之前,尽管 Java 语言中接口已经非常优秀了,但相比其他面向对象的语言而言,Java 接口存在如下不足之处。

(1) 不能可选实现方法。接口的方法全部是抽象的,实现接口时必须全部实现接口中方法,哪怕是有些方法并不需要,也必须实现。

（2）没有静态方法。针对这些问题，Java 8 在接口中提供了声明默认方法和静态方法的能力。接口示例代码如下：

```
//InterfaceA.java 文件
package com.a51work6;

public interface InterfaceA {

    void methodA();

    String methodB();

    //默认方法
    default int methodC() {
        return 0;
    }

    //默认方法
    default String methodD() {
        return "这是默认方法…";
    }

    //静态方法
    static double methodE() {
        return 0.0;
    }
}
```

在接口 InterfaceA 中声明了两个抽象方法 methodA 和 methodB，两个默认方法 methodC 和 methodD，还声明了静态方法 methodE。接口中的默认方法类似于类中的具体方法，给出了具体实现，只是方法修饰符是 default。接口中静态方法类似于类中静态方法。

实现接口示例代码如下：

```
//ABC.java 文件
package com.a51work6.imp;

import com.a51work6.InterfaceA;

public class ABC implements InterfaceA {

    @Override
    public void methodA() {
    }

    @Override
    public String methodB() {
        return "实现 methodB 方法…";
    }

    @Override
    public int methodC() {
        return 500;
    }
}
```

实现接口时接口中原有的抽象方法在实现类中必须实现。默认方法可以根据需要有选择地实现(覆盖)。静态方法不需要实现,实现类中不能拥有接口中的静态方法。

上述代码中 ABC 类实现了 InterfaceA 接口,InterfaceA 接口中的两个默认方法 ABC 只是实现(覆盖)了 methodC。

调用代码如下:

```
//HelloWorld.java 文件
package com.a51work6.imp;

import com.a51work6.InterfaceA;

public class HelloWorld {

    public static void main(String[] args) {

        //声明接口类型,对象是实现类,发生多态
        InterfaceA abc = new ABC();

        //访问实现类 methodB 方法
        System.out.println(abc.methodB());

        //访问默认方法 methodC
        System.out.println(abc.methodC());                ①

        //访问默认方法 methodD
        System.out.println(abc.methodD());                ②

        //访问 InterfaceA 静态方法 methodE
        System.out.println(InterfaceA.methodE());         ③
    }
}
```

运行结果如下:

```
实现 methodB 方法…
500
这是默认方法…
0.0
```

从运行结果可见,代码第①行调用默认方法 methodC,是调用类 ABC 中的实现;代码第②行调用默认方法 methodD,是调用接口 InterfaceA 中的实现;代码第③行调用接口静态方法,只能通过接口名(InterfaceA)调用,不能通过实现类 ABC 调用,可以这样理解接口中声明的静态方法:与其他实现类没有任何关系。

13.3 抽象类与接口的区别

经过前面的学习,应该对于抽象类和接口有所了解,但可能会有这样的疑问:抽象类和接口有什么区别?本节就回答这个问题。

抽象类与接口区别如下。

(1) 接口支持多继承,而抽象类(包括具体类)只能继承一个父类。

(2) 接口中不能有实例成员变量,接口所声明的成员变量全部是静态常量,即便变量不加 public static final 修饰符也是静态常量。抽象类与普通类一样,各种形式的成员变量都可以声明。

(3) 接口中没有包含构造方法。由于没有实例成员变量,也就不需要构造方法。抽象类中可以有实例成员变量,也需要构造方法。

(4) 抽象类中可以声明抽象方法和具体方法。Java 8 之前接口中只有抽象方法,而 Java 8 之后接口中也可以声明具体方法,具体方法通过声明默认方法实现。

提示　学习了接口默认方法后,有些读者还会有这样的疑问,Java 8 之后接口可以声明抽象方法和具体方法,这就相当于与抽象类一样了吗?在多数情况下接口不能替代抽象类,例如,当需要维护一个对象的信息和状态时只能使用抽象类,而接口不行,因为维护一个对象的信息和状态需要存储在实例成员变量中,而接口中不能声明实例成员变量。

本章小结

通过对本章的学习,读者可以了解抽象类和接口的概念,掌握如何声明抽象类和接口,如何实现抽象类和接口;了解 Java 8 之后接口的新变化;熟悉抽象类和接口的区别。

13.4　同步练习

1. 关于接口的定义和实现,以下描述正确的是(　　)。
   A. 接口定义中的方法都只有定义没有实现
   B. 接口定义中的变量都必须写明 final 和 static
   C. 如果一个接口由多个类来实现,则这些类在实现该接口中的方法时应采用统一的代码
   D. 如果一个类实现一个接口,则必须实现该接口中的所有方法

2. 判断对错。

(1) abstract 是抽象修饰符,可以用来修饰类和方法。(　　)

(2) Java 语言中的接口可以继承,一个接口通过关键字 extends 可以继承另一个接口。(　　)

3. 下列选项中,用于定义接口的关键字是(　　)。
   A. import　　B. implements　　C. interface　　D. protected

第 14 章 Java 常用类

CHAPTER 14

在 Java SE 中提供了众多的类和接口，其中很多类前面已经使用过，如 String、StringBuilder 和 StringBuffer 等。由于数量众多，本书不可能一一介绍，也没有这个必要。本章归纳了 Java 中一些在日常开发过程中常用的类，至于其他不常用的类，可以查阅 Java SE API 文档。

14.1 Java 根类——Object

第一个应该介绍的常用类就是 java. lang. Object 类，它是 Java 所有类的根类，Java 所有类都直接或间接继承自 Object 类，它是所有类的“祖先”。Object 类属于 java. lang 包中的类型，不需要显式使用 import 语句引入，它是由解释器自动引入。

Object 类有很多方法，常用的方法如下。

- String toString()：返回该对象的字符串表示。
- boolean equals(Object obj)：指示其他某个对象是否与此对象“相等”。

这些方法都是需要在子类中用来覆盖的，下面详细解释它们的用法。

14.1.1 toString()方法

为了日志输出等处理方便，所有的对象都可以以文本方式表示，需要在该对象所在类中覆盖 toString()方法。如果没有覆盖 toString()方法，默认的字符串是“类名@对象的十六进制哈希码[①]”。

下面看一个示例。在前面章节介绍过 Person 类，它的代码如下：

```
//Person.java 文件
package com.a51work6;

public class Person {

    String name;
    int age;
```

① 哈希码(hashCode)，每个 Java 对象都有哈希码属性，哈希码可以用来标识对象，提高对象在集合操作中的执行效率。

```
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {                          ①
        return "Person [name=" + name + ", age=" + age + "]";
    }

}
```

上述代码第①行覆盖 toString()方法,返回什么样的字符串完全是自定义的,只要能够表示当前类和当前对象即可,本例是将 Person 成员变量拼接成为一个字符串。

调用代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Person person = new Person("Tony", 18);
        //打印过程自动调用 person 的 toString()方法
        System.out.println(person);
    }
}
```

输出结果如下:

```
Person [name=Tony, age=18]
```

使用 System.out.println 等输出语句可以自动调用对象的 toString()方法,将对象转换为字符串输出。读者可以测试一下,如果 Person 中没有覆盖 toString()方法会是什么样子?它会输出类似如下的字符串:

```
com.a51work6.Person@15db9742
```

14.1.2 对象比较方法

在前面学习字符串比较时,曾经介绍过有两种比较方法:==运算符和 equals()方法,==运算符是比较两个引用变量是否指向同一个实例,equals()方法是比较两个对象的内容是否相等,通常字符串的比较只是关心其内容是否相等。

事实上 equals()方法是继承自 Object 的,所有对象都可以通过 equals()方法比较,问题是比较的规则是什么,例如两个人(Person 对象)相等是指什么?是名字?是年龄?问题的关键是需要指定相等的规则,就是要指定比较的是哪些属性相等,所以为了比较两个

Person 对象相等，需要覆盖 equals()方法，在该方法中指定比较规则。

修改 Person 代码如下：

```
//Person.java 文件
package com.a51work6;
public class Person {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }

    @Override
    public boolean equals(Object otherObject) {                ①

        //判断比较的参数也是 Person 类型
        if (otherObject instanceof Person) {                   ②
            Person otherPerson = (Person) otherObject;         ③
            //年龄作为比较规则
            if (this.age == otherPerson.age) {                 ④
                return true;
            }
        }
        return false;
    }

}
```

上述代码第①行覆盖了 equals()，为了防止传入的参数对象不是 Person 类型，需要使用 instanceof 运算符判断一下，见代码第②行。如果是 Person 类型，通过代码第③行强制类型转换为 Person。代码第④行进行比较，把年龄作为比较是否相等的规则，不管其他属性，只要是年龄相等，则认为两个 Person 对象相等。

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
```

```
        Person person1 = new Person("Tony", 20);
        Person person2 = new Person("Tom", 20);

        System.out.println(person1 == person2);          //false
        System.out.println(person1.equals(person2));     //true
    }
}
```

上述代码中创建了两个 Person 对象，它们具有相关的年龄，这两个 Person 对象使用==比较结果是 false，因为它们是两个不同的对象；使用 equals()方法比较结果是 true。

14.2 包装类

在 Java 中 8 种基本数据类型不属于类，不具备“对象”的特征，没有成员变量和方法，不方便进行面向对象的操作。为此，Java 提供包装类(Wrapper Class)来将基本数据类型包装成类，每个 Java 基本数据类型在 java.lang 包中都有一个相应的包装类，每个包装类对象封装一个基本数据类型数值。对应关系如表 14-1 所示，除 int 和 char 类型外，其他的类型对应规则就是第一个字母大写。

表 14-1 基本数据类型与包装类对应关系

基本数据类型	包 装 类	基本数据类型	包 装 类
boolean	Boolean	int	Integer
byte	Byte	long	Long
char	Character	float	Float
short	Short	double	Double

包装类都是 final 的，不能被继承。包装类都是不可变类，类似于 String 类，一旦创建了对象，其内容就不可以修改。包装类还可以分成 3 种不同类别：数值包装类、Character 和 Boolean。下面分别详细介绍。

14.2.1 数值包装类

数值包装类包括 Byte、Short、Integer、Long、Float 和 Double，它们都有一些相同的特点。

1. 构造方法类似

每一个数值包装类都有两个构造方法。以 Integer 为例，Integer 构造方法如下。

- Integer(int value)：通过指定一个数值构造 Integer 对象。
- Integer(String s)：通过指定一个字符串 s 构造对象，s 是十进制字符串表示的数值。

2. 共同的父类

这 6 个数值包装类有一个共同的父类——Number，Number 是一个抽象类，除了这 6 个子类，还有 AtomicInteger、AtomicLong、BigDecimal 和 BigInteger，其中 BigDecimal 和 BigInteger 后面还会详细介绍。Number 是抽象类，要求它的子类必须实现如下 6 个方法。

- byte byteValue()：将当前包装的对象转换为 byte 类型的数值。
- double doubleValue()：将当前包装的对象转换为 double 类型的数值。
- float floatValue()：将当前包装的对象转换为 float 类型的数值。
- int intValue()：将当前包装的对象转换为 int 类型的数值。
- long longValue()：将当前包装的对象转换为 long 类型的数值。
- short shortValue()：将当前包装的对象转换为 short 类型的数值。

通过这 6 个方法，数值包装类可以互相转换这 6 种数值，但是需要注意的是大范围数值转换为小范围的数值，如果数值本身很大，可以会导致数值精度的丢失。

3. compareTo()方法

每一个数值包装类都有 int compareTo(数值包装类对象)方法，可以进行包装对象的比较。方法返回值是 int。如果返回值是 0，则相等；如果返回值小于 0，则此对象小于参数对象；如果返回值大于 0，则此对象大于参数对象。

4. 字符串转换为基本数据类型

每一个数值包装类都提供一些静态 parseXXX()方法将字符串转换为对应的基本数据类型。以 Integer 为例，方法定义如下。

- static int parseInt(String s)：将字符串 s 转换为有符号的十进制整数。
- static int parseInt(String s，int radix)：将字符串 s 转换为有符号的整数，radix 是基数，基数用来指定进制。注意，这种指定基数的方法在浮点数包装类(Double 和 Float)中是没有的。

5. 基本数据类型转换为字符串

每一个数值包装类都提供一些静态 toString()方法实现将基本数据类型数值转换为字符串。以 Integer 为例，方法定义如下。

- static String toString(int i)：将该整数 i 转换为有符号的十进制表示的字符串。
- static String toString(int i，int radix)：将该整数 i 转换为有符号的特定进制表示的字符串，radix 是基数，可以指定进制。注意，这种指定基数的方法在浮点数包装类(Double 和 Float)中是没有的。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //1. 构造方法
        //创建数值为 80 的 Integer 对象
        Integer objInt = new Integer(80);
        //创建数值为 80.0 的 Double 对象
        Double objDouble = new Double(80.0);
        //通过"80.0"字符串创建数值为 80.0 的 Float 对象
        Float objFloat = new Float("80.0");
        //通过"80"字符串创建数值为 80 的 Long 对象
```

```
        Long objLong = new Long("80");

        //2. Number 类方法
        //Integer 对象转换为 long 数值
        long longVar = objInt.longValue();
        //Double 对象转换为 int 数值
        int intVar = objDouble.intValue();
        System.out.println("intVar = " + intVar);
        System.out.println("longVar = " + longVar);

        //3. compareTo()方法
        Float objFloat2 = new Float(100);
        int result = objFloat.compareTo(objFloat2);
        //result = -1,表示 objFloat 小于 objFloat2
        System.out.println(result);

        //4. 字符串转换为基本数据类型
        //十进制"100"字符串转换为十进制数为 100
        int intVar2 = Integer.parseInt("100");
        //十六进制"ABC"字符串转换为十六进制数为 2748
        int intVar3 = Integer.parseInt("ABC", 16);
        System.out.println("intVar2 = " + intVar2);
        System.out.println("intVar3 = " + intVar3);

        //5. 基本数据类型转换为字符串
        //100 转换为十进制字符串
        String str1 = Integer.toString(100);
        //100 转换为十六进制字符串,结果是 64
        String str2 = Integer.toString(100, 16);
        System.out.println("str1 = " + str1);
        System.out.println("str2 = " + str2);

    }
}
```

代码中注释比较清楚,这里不再解释了。

14.2.2 Character 类

Character 类是 char 类型的包装类。Character 类常用方法如下。

- Character(char value):构造方法,通过 char 值创建一个新的 Character 对象。
- char charValue():返回此 Character 对象的值。
- int compareTo(Character anotherCharacter):方法返回值是 int。如果返回值是 0,则相等;如果返回值小于 0,则此对象小于参数对象;如果返回值大于 0,则此对象大于参数对象。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //创建数值为'A'的 Character 对象
        Character objChar1 = new Character('A');
        //从 Character 对象返回 char 值
        char ch = objChar1.charValue();

        //字符比较
        Character objChar2 = new Character('C');
        int result = objChar1.compareTo(objChar2);
        //result = -2,表示 objChar1 小于 objChar2
        if (result < 0) {
            System.out.println("objChar1 小于 objChar2");
        }
    }
}
```

14.2.3 Boolean 类

Boolean 类是 boolean 类型的包装类。

1. 构造方法

Boolean 类有两个构造方法，构造方法定义如下。

- Boolean(boolean value)：通过一个 boolean 值创建 Boolean 对象。
- Boolean(String s)：通过字符串创建 Boolean 对象。s 不能为 null，s 如果是忽略大小写"true"，则转换为 true 对象，其他字符串都转换为 false 对象。

2. compareTo()方法

Boolean 类有 int compareTo(Boolean 包装类对象)方法，可以进行包装对象的比较。方法返回值是 int。如果返回值是 0，则相等；如果返回值小于 0，则此对象小于参数对象；如果返回值大于 0，则此对象大于参数对象。

3. 字符串转换为 boolean 类型

Boolean 包装类都提供静态 parseBoolean()方法实现将字符串转换为对应的 boolean 类型，方法定义如下：

```
static boolean parseBoolean(String s)
```

将字符串转换为对应的 boolean 类。s 不能为 null，s 如果是忽略大小写"true"，则转换为 true，其他字符串都转换为 false。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //创建数值为 true 的 Character 对象 true
        Boolean obj1 = new Boolean(true);
        //通过字符串"true"创建 Character 对象 true
        Boolean obj2 = new Boolean("true");
        //通过字符串"True"创建 Character 对象 true
        Boolean obj3 = new Boolean("True");
        //通过字符串"TRUE"创建 Character 对象 true
        Boolean obj4 = new Boolean("TRUE");
        //通过字符串"false"创建 Character 对象 false
        Boolean obj5 = new Boolean("false");
        //通过字符串"Yes"创建 Character 对象 false
        Boolean obj6 = new Boolean("Yes");
        //通过字符串"abc"创建 Character 对象 false
        Boolean obj7 = new Boolean("abc");


        boolean b1 = Boolean.parseBoolean("true");
        boolean b2 = Boolean.parseBoolean("True");
        boolean b3 = Boolean.parseBoolean("TRUE");
        boolean b4 = Boolean.parseBoolean("false");
        boolean b5 = Boolean.parseBoolean("Yes");
        boolean b6 = Boolean.parseBoolean("abc");
        …
    }
}
```

14.2.4 自动装箱/拆箱

包装类丰富了 Java 语言面向对象，提供了原来基本数据类型没有的方法。但是也带来了使用的不便。例如，如下代码试图对包装类对象进行算数运算，在 Java 5 之前代码第①行会发生编译错误，想想可以理解，这些对象不能简单使用算数运算符连接起来。

```
//创建数值为 80 的 Integer 对象
Integer objInt = new Integer(80);
//创建数值为 80.0 的 Double 对象
Double objDouble = new Double(80.0);
//算数运算
double sum = objInt + objDouble;        //Java 5 之前有编译错误        ①
```

但是代码第①行在 Java 5 之后可以编译通过了，并能计算出正确的结果。这是因为 Java 5 之后提供了拆箱(unboxing)功能，拆箱能够将包装类对象自动转换为基本数据类型

的数值,而不需要使用 intValue()或 doubleValue()等方法。类似 Java 5 还提供了相反功能——自动装箱(autoboxing),装箱能够自动地将基本数据类型的数值自动转换为包装类对象,而不需要使用构造方法。

示例代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Integer objInt = new Integer(80);
        Double objDouble = new Double(80.0);
        //自动拆箱
        double sum = objInt + objDouble;

        //自动装箱
        //自动装箱'C'转换为 Character 对象
        Character objChar = 'C';
        //自动装箱 true 转换为 Boolean 对象
        Boolean objBoolean = true;
        //自动装箱 80.0f 转换为 Float 对象
        Float objFloat = 80.0f;

        //自动装箱 100 转换为 Integer 对象
        display(100);

        //避免出现下面的情况
        Integer obj = null;                                          ①
        int intVar = obj;          //运行期异常 NullPointerException  ②

    }

    /**
     * @param objInt Integer 对象
     * @return int 数值
     */
    public static int display(Integer objInt) {

        System.out.println(objInt);

        //return objInt.intValue();
        //自动拆箱 Integer 对象转换为 int
        return objInt;
    }
}
```

在自动装箱和拆箱时,要避免空对象,代码第①行 obj 是 null,代码第②行会发生运行期 NullPointerException 异常,这是因为拆箱的过程本质上是调用 intValue()方法实现的,试图访问空对象的方法和成员变量,就会抛出运行期 NullPointerException 异常。

14.3 Math 类

Java 语言是彻底的面向对象语言，哪怕是进行数学运算也封装到一个类中，这个类是 java.lang.Math，Math 类是 final 的，不能被继承。Math 类中包含用于进行基本数学运算的方法，如指数、对数、平方根和三角函数等。

1. 舍入方法

- static double ceil(double a)：返回大于或等于 a 的最小整数。
- static double floor(double a)：返回小于或等于 a 的最大整数。
- static int round(float a)：四舍五入方法。

2. 最大值和最小值

- static int min(int a,int b)：取两个 int 整数中较小的一个整数。
- static int min(long a,long b)：取两个 long 整数中较小的一个整数。
- static int min(float a,float b)：取两个 float 浮点数中较小的一个浮点数。
- static int min(double a,double b)：取两个 double 浮点数中较小的一个浮点数。

max 方法取两个数中较大的一个数，max 方法与 min 方法参数类似也有 4 个版本，这里不再赘述。

3. 绝对值

- static int abs(int a)：取 int 整数 a 的绝对值。
- static long abs(long a)：取 long 整数 a 的绝对值。
- static float abs(float a)：取 float 浮点数 a 的绝对值。
- static double abs(double a)：取 double 浮点数 a 的绝对值。

4. 三角函数

- static double sin(double a)：返回角的三角正弦。
- static double cos(double a)：返回角的三角余弦。
- static double tan(double a)：返回角的三角正切。
- static double asin(double a)：返回一个值的反正弦。
- static double acos(double a)：返回一个值的反余弦。
- static double atan(double a)：返回一个值的反正切。
- static double toDegrees(double angrad)：将弧度转换为角度。
- static double toRadians(double angdeg)：将角度转换为弧度。

5. 对数运算

static double log(double a)：返回 a 的自然对数。

6. 平方根

static double sqrt(double a)：返回 a 的正平方根。

7. 幂运算

static double pow(double a,double b)：返回第一个参数的第二个参数次幂的值。

8. 计算随机值

static double random()：返回大于或等于 0.0 且小于 1.0 的随机数。

9. 常量

- 圆周率 PI。
- 自然对数的底数 E。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        double[] nums = { 1.4, 1.5, 1.6 };

        //测试最大值和最小值
        System.out.printf("min(%.1f, %.1f) = %.1f\n", nums[1], nums[2],
                Math.min(nums[1], nums[2]));
        System.out.printf("max(%.1f, %.1f) = %.1f\n", nums[1], nums[2],
                Math.max(nums[1], nums[2]));
        System.out.println();

        //测试三角函数
        //1π 弧度 = 180°
        System.out.printf("toDegrees(0.5π) = %f\n", Math.toDegrees(0.5 * Math.PI));
        System.out.printf("toRadians(180/π) = %f\n", Math.toRadians(180 / Math.PI));
        System.out.println();

        //测试平方根
        System.out.printf("sqrt(%.1f) = %f\n", nums[2], Math.sqrt(nums[2]));
        System.out.println();

        //测试幂运算
        System.out.printf("pow(8, 3) = %f\n", Math.pow(8, 3));
        System.out.println();

        //测试计算随机值
        System.out.printf("0.0~1.0 之间的随机数 = %f\n", Math.random());
        System.out.println();

        //测试舍入
        for (double num : nums) {
            display(num);
        }

    }

    //测试舍入方法
    public static void display(double n) {
        System.out.printf("ceil(%.1f) = %.1f\n", n, Math.ceil(n));
```

```
        System.out.printf("floor(%.1f) = %.1f\n", n, Math.floor(n));
        System.out.printf("round(%.1f) = %d\n", n, Math.round(n));
        System.out.println();
    }
}
```

运行结果如下：

```
min(1.5, 1.6) = 1.5
max(1.5, 1.6) = 1.6

toDegrees(0.5π) = 90.000000
toRadians(180/π) = 1.000000

sqrt(1.6) = 1.264911

pow(8, 3) = 512.000000

0.0～1.0 之间的随机数 = 0.881115

ceil(1.4)  = 2.0
floor(1.4) = 1.0
round(1.4) = 1

ceil(1.5)  = 2.0
floor(1.5) = 1.0
round(1.5) = 2

ceil(1.6)  = 2.0
floor(1.6) = 1.0
round(1.6) = 2
```

上述代码比较简单，这里不再赘述。

14.4 大数值

对货币等大数值数据进行计算时，int、long、float 和 double 等基本数据类型已经在精度方面不能满足需求了。为此 Java 提供了两个大数值类：BigInteger 和 BigDecimal，这两个类都继承自 Number 抽象类。

14.4.1 BigInteger

java.math.BigInteger 是不可变的任意精度的大整数。BigInteger 构造方法有很多，其中字符串参数的构造方法有两个。

- BigInteger(String val)：将十进制字符串 val 转换为 BigInteger 对象。
- BigInteger(String val, int radix)：按照指定基数 radix 将字符串 val 转换为 BigInteger 对象。

BigInteger 提供多种方法，下面列举几个常用的方法。

- int compareTo(BigInteger val)：将当前对象与参数 val 进行比较，方法返回值是 int。如果返回值是 0，则相等；如果返回值小于 0，则此对象小于参数对象；如果返回值大于 0，则此对象大于参数对象。
- BigInteger add(BigInteger val)：加运算，当前对象数值加参数 val。
- BigInteger subtract(BigInteger val)：减运算，当前对象数值减参数 val。
- BigInteger multiply(BigInteger val)：乘运算，当前对象数值乘参数 val。
- BigInteger divide(BigInteger val)：除运算，当前对象数值除以参数 val。

另外，BigInteger 继承了抽象类 Number，所以它还实现抽象类 Number 的 6 个方法，具体方法参考 14.2.1 节。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.math.BigInteger;

public class HelloWorld {

    public static void main(String[] args) {

        //创建 BigInteger,字符串表示十进制数值
        BigInteger number1 = new BigInteger("999999999999");
        //创建 BigInteger,字符串表示十六进制数值
        BigInteger number2 = new BigInteger("567800000", 16);

        //加法操作
        System.out.println("加法操作:" + number1.add(number2));
        //减法操作
        System.out.println("减法操作:" + number1.subtract(number2));
        //乘法操作
        System.out.println("乘法操作:" + number1.multiply(number2));
        //除法操作
        System.out.println("除法操作:" + number1.divide(number2));
    }
}
```

运行结果如下：

```
加法操作:1023211278335
减法操作:976788721663
乘法操作:23211278335976788721664
除法操作:43
```

上述代码比较简单，这里不再赘述。

14.4.2 BigDecimal

java.math.BigDecimal 是不可变的任意精度的有符号十进制数。BigDecimal 构造方法有很多，下面列举几个常用的构造方法。

- BigDecimal(BigInteger val)：将 BigInteger 对象 val 转换为 BigDecimal 对象。
- BigDecimal(double val)：将 double 转换为 BigDecimal 对象，参数 val 是 double 类型的二进制浮点值准确的十进制表示形式。
- BigDecimal(int val)：将 int 转换为 BigDecimal 对象。
- BigDecimal(long val)：将 long 转换为 BigDecimal 对象。
- BigDecimal(String val)：将字符串表示数值形式转换为 BigDecimal 对象。

BigDecimal 提供多种方法，下面列举几个常用的方法。

- int compareTo(BigDecimal val)：将当前对象与参数 val 进行比较，方法返回值是 int。如果返回值是 0，则相等；如果返回值小于 0，则此对象小于参数对象；如果返回值大于 0，则此对象大于参数对象。
- BigDecimal add(BigDecimal val)：加运算，当前对象数值加参数 val。
- BigDecimal subtract(BigDecimal val)：减运算，当前对象数值减参数 val。
- BigDecimal multiply(BigDecimal val)：乘运算，当前对象数值乘参数 val。
- BigDecimal divide(BigDecimal val)：除运算，当前对象数值除以参数 val。
- BigDecimal divide(BigDecimal val, int roundingMode)：除运算，当前对象数值除以参数 val。roundingMode 为要应用的舍入模式。

另外，BigDecimal 继承了抽象类 Number，所以它还实现抽象类 Number 的 6 个方法，具体方法参考 14.2.1 节。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.math.BigDecimal;

public class HelloWorld {

    public static void main(String[] args) {

        //创建 BigDecimal,通过字符串参数创建
        BigDecimal number1 = new BigDecimal("999999999.99988888");
        //创建 BigDecimal,通过 double 参数创建
        BigDecimal number2 = new BigDecimal(567800000.888888);

        //加法操作
        System.out.println("加法操作:" + number1.add(number2));
        //减法操作
        System.out.println("减法操作:" + number1.subtract(number2));
        //乘法操作
        System.out.println("乘法操作:" + number1.multiply(number2));
```

```
            //除法操作
            System.out.println("除法操作:"
                        + number1.divide(number2, BigDecimal.ROUND_HALF_UP));   ①
        }
    }
```

运行结果如下：

```
加法操作:1567800000.888776881441955566406 25
减法操作:432199999.11100087855804443359375
乘法操作:567800000888824907.50585679317152976989746093750 00
除法操作:1.76118351
```

上述代码第①行是进行除法运算，该方法需要指定舍入模式，如果不指定舍入模式，则会发生运行期异常 ArithmeticException，舍入模式 BigDecimal.ROUND_HALF_UP 是四舍五入。

14.5 日期时间相关类

Java 8 之前日期类是 java.util.Date，Date 类比较古老，其中的很多方法现在已经废弃，但是目前仍然有很多程序还在使用 Date 类，据此，本节介绍一下 Date 类及日期时间相关类的使用。

此外，Java 8 之前与日期时间相关的类还有 DateFormat、Calendar 和 TimeZone，DateFormat 用于日期格式化，Calendar 为日历类，TimeZone 是时区类。

提示 在 Java SE 核心类中有两个 Date，分别是 java.util.Date 和 java.sql.Date。java.util.Date 就是本节要介绍的日期时间类，而 java.sql.Date 是 JDBC 中日期字段类型。

14.5.1 Date 类

Date 类中有很多构造方法和普通方法。Date 类的构造方法如下。

- Date()：用当前时间创建 Date 对象，精确到毫秒。
- Date(long date)：指定标准基准时间以来的毫秒数创建 Date 对象。标准基准时间是格林尼治时间 1970 年 1 月 1 日 00:00:00。

Date 类的普通方法如下。

- boolean after(Date when)：测试此日期是否在 when 日期之后。
- boolean before(Date when)：测试此日期是否在 when 日期之前。
- int compareTo(Date anotherDate)：比较两个日期的顺序。如果参数日期等于此日期，则返回值 0；如果此日期在参数日期之前，则返回小于 0 的值；如果此日期在参数日期之后，则返回大于 0 的值。
- long getTime()：返回自 1970 年 1 月 1 日 00:00:00 以来此 Date 对象表示的毫秒数。
- void setTime(long time)：用毫秒数 time 设置日期对象，time 是自 1970 年 1 月 1 日 00:00:00 以来此 Date 对象表示的毫秒数。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) {

        Date now = new Date();                                          ①
        System.out.println("now = " + now);                             ②
        System.out.println("now.getTime() = " + now.getTime());
        System.out.println();

        Date date = new Date(1234567890123L);                           ③
        System.out.println("date = " + date);

        //测试 now 和 date 日期
        display(now, date);                                             ④

        //重新设置日期 time
        date.setTime(9999999999999L);                                   ⑤

        System.out.println("修改之后的 date = " + date);

        //重新测试 now 和 date 日期
        display(now, date);                                             ⑥

    }

    //测试 after、before 和 compareTo 方法
    public static void display(Date now, Date date) {
        System.out.println();
        System.out.println("now.after(date) = " + now.after(date));
        System.out.println("now.before(date) = " + now.before(date));
        System.out.println("now.compareTo(date) = " + now.compareTo(date));
        System.out.println();
    }
}
```

运行结果如下：

```
now = Sun Jun 04 10:03:09 CST 2017
now.getTime() = 1496541789730

date = Sat Feb 14 07:31:30 CST 2009

now.after(date) = true
```

```
now.before(date) = false
now.compareTo(date) = 1

修改之后的 date = Sun Nov 21 01:46:39 CST 2286

now.after(date) = false
now.before(date) = true
now.compareTo(date) = -1
```

上述代码①行是创建当前日期对象，代码第②行是打印输出当前日期对象，从输出结果可见是 Sun Jun 04 10:03:09 CST 2017，其中 CST 是美国中部标准时间。

代码第③行是通过 long 整数 1234567890123L 创建日期对象，打印输出 date 日期是 Sat Feb 14 07:31:30 CST 2009。代码第⑤行又重新设置了 time，之后打印输出 date 日期是 Sun Nov 21 01:46:39 CST 2286。

代码第④行和第⑥行两次调用 display 方法测试 after、before 和 compareTo 方法。

14.5.2 日期格式化和解析

上一节示例日期输出结果，如 Sun Jun 04 10：03：09 CST 2017，这个时间不符合中国人的习惯，此时需要对日期进行格式化输出。日期格式化类是 java.text.DateFormat，DateFormat 是抽象类，它的常用具体类是 java.text.SimpleDateFormat。

DateFormat 中提供日期格式化和日期解析方法，具体方法说明如下。

- String format(Date date)：将一个 Date 格式化为日期/时间字符串。
- Date parse(String source)：从给定字符串的开始解析文本，以生成一个日期对象。如果解析失败，则抛出 ParseException。

另外，具体类 SimpleDateFormat 构造方法如下。

- SimpleDateFormat()：用默认的模式和默认语言环境的日期格式符号构造 SimpleDateFormat。
- SimpleDateFormat(String pattern)：用给定的模式和默认语言环境的日期格式符号构造 SimpleDateFormat。pattern 参数是日期和时间格式模式。

表 14-2 所示是常用的日期和时间格式。

表 14-2 常用的日期和时间格式

字 母	日期或时间元素	字 母	日期或时间元素
y	年	a	AM/PM 标记
M	年中的月份	m	小时中的分钟数
D	年中的天数	s	分钟中的秒数
d	月份中的天数	S	毫秒数
H	一天中的小时数(0～23)	Z	时区
h	AM/PM 中的小时数(1～12)		

示例代码如下：

```
//HelloWorld.java文件
package com.a51work6;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) throws ParseException {              ①

        Date date = new Date(1234567890123L);                                   ②
        System.out.println("格式化前 date = " + date);

        DateFormat df = new SimpleDateFormat();                                 ③
        System.out.println("格式化后 date = " + df.format(date));               ④
        df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");                       ⑤
        System.out.println("格式化后 date = " + df.format(date));               ⑥

        String dateString = "2018-08-18 08:18:58";
        Date date1 = df.parse(dateString);                                      ⑦
        System.out.println("从字符串获得日期对象 = " + date1);
    }
}
```

运行结果如下：

```
格式化前 date = Sat Feb 14 07:31:30 CST 2009
格式化后 date = 09-2-14 上午 7:31
格式化后 date = 2009-02-14 07:31:30
从字符串获得日期对象 = Sat Aug 18 08:18:58 CST 2018
```

上述代码第②行创建日期对象；代码第③行采用默认构造方法创建日期格式化SimpleDateFormat对象；代码第④行进行格式化输出，结果是“09-2-14 上午 7:31”，这个格式化采用的是当前操作系统默认格式，在实际开发时用得不多。代码第⑤行重新创建SimpleDateFormat对象，这里指定它的日期时间格式为"yyyy-MM-dd HH:mm:ss"；代码第⑥行是格式化输出，结果是“2009-02-14 07:31:30”，开发人员还可以根据自己的需要设置其他格式。

日期格式化，一方面可以将日期对象转换为特定格式的字符串；另一方面可以将特定格式的字符串转换为日期对象。代码第⑦行是将字符串"2018-08-18 08:18:58"转换为日期对象。

注意 并不是所有的字符串都能够转换为日期，如果转换失败，parse方法会抛出异常ParseException。由于ParseException异常是受检查类型异常，这种异常必须处理，本例是抛出处理，见代码第①行main方法后的throws ParseException语句。目前只需了解

异常这样处理就可以了,异常将在第17章详细说明。

14.5.3 Calendar 类

有时为了取得更多的日期时间信息,或对日期时间进行操作,可以使用 java.util.Calendar 类。Calendar 是一个抽象类,不能实例化,但是通过静态工厂方法 getInstance() 获得 Calendar 实例。

Calendar 类的主要方法如下。

- static Calendar getInstance():使用默认时区和语言环境获得一个日历。
- void set(int field,int value):将给定的日历字段设置为给定值。
- void set(int year,int month,int date):设置日历字段 YEAR、MONTH 和 DAY_OF_MONTH 的值。
- Date getTime():返回一个表示此 Calendar 时间值(从 1970 年 1 月 1 日 00:00:00 至现在的毫秒数)的 Date 对象。
- boolean after(Object when):判断此 Calendar 表示的时间是否在指定时间之后,返回判断结果。
- boolean before(Object when):判断此 Calendar 表示的时间是否在指定时间之前,返回判断结果。
- int compareTo(Calendar anotherCalendar):比较两个 Calendar 对象表示的时间值。

日历示例代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class HelloWorld {

    public static void main(String[] args) throws ParseException {

        //获得默认的日历对象
        Calendar calendar = Calendar.getInstance();
        //设置日期 2018 年 8 月 18 日
        calendar.set(2018, 7, 18);                        ①

        //通过日历获得 Date 对象
        Date date = calendar.getTime();                   ②
        System.out.println("格式化前 date = " + date);
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        System.out.println("格式化后 date = " + df.format(date));
        System.out.println();
```

```
        calendar.clear();                    ③
        //设置日期 2018 年 8 月 28 日
        calendar.set(Calendar.YEAR, 2018);   ④
        calendar.set(Calendar.MONTH, 7);     ⑤
        calendar.set(Calendar.DATE, 28);     ⑥

        //通过日历获得 Date 对象
        date = calendar.getTime();
        System.out.println("格式化前 date = " + date);
        System.out.println("格式化后 date = " + df.format(date));
    }
}
```

运行结果如下：

```
格式化前 date = Sat Aug 18 14:47:22 CST 2018
格式化后 date = 2018-08-18

格式化前 date = Tue Aug 28 00:00:00 CST 2018
格式化后 date = 2018-08-28
```

上述代码第①行是设置日历的年、月和日字段，注意在设置“月”时，应该是“月份－1”，因为日历中的月份中第一个月是 0，第二个月是 1，以此类推，本例中设置 8 月份则实际参数应该为 7。代码第②行是通过日历获得日期对象。

代码第③行 calendar.clear()语句是重新初始化日历对象。代码第④～⑥行分别设置日历的年、月和日字段。

14.6 Java 8 新日期时间相关类

Java 8 之后提供了新的日期时间相关类、接口和枚举，这些类型内容非常多，令人生畏。但是使用起来非常方便。

14.6.1 时间和日期

Java 8 之后提供了新的日期时间类有 3 个：LocalDate、LocalTime 和 LocalDateTime，它们都位于 java.time 包中，LocalDate 表示一个不可变的日期对象；LocalTime 表示一个不可变的时间对象；LocalDateTime 表示一个不可变的日期和时间。

这 3 个类有类似的方法，先看看创建日期时间对象相关方法。这 3 个类并没有提供公有的构造方法，创建它们的对象可以使用静态工厂方法，主要有 now()和 of()方法。

now()方法说明如下。

- static LocalDate now()：LocalDate 静态工厂方法，该方法使用默认时区获得当前日期，返回 LocalDate 对象。
- static LocalTime now()：LocalTime 静态工厂方法，该方法使用默认时区获得当前时间，返回 LocalTime 对象。
- static LocalDateTime now()：LocalDateTime 静态工厂方法，该方法使用默认时区

获得当前日期时间，返回 LocalDateTime 对象。

of()方法有很多重载方法，说明如下。

- static LocalDateTime of(int year,int month,int dayOfMonth,int hour,int minute,int second)：按照指定的年、月、日、时、分和秒获得 LocalDateTime 实例，将纳秒设置为零。
- static LocalTime of(int hour,int minute,int second)：按照指定的时、分和秒获取一个 LocalTime 实例。
- static LocalDate of(int year,int month,int dayOfMonth)：按照指定的年、月和日获得一个 LocalDate 实例，日期中年、月和日必须有效，否则将抛出异常。

上述方法中的参数取值范围如表 14-3 所示。

表 14-3　参数取值范围

参　　数	说　　明
year	从－999 999 999 到 999 999 999 的年份
month	一年中的月份，从 1 到 12
dayOfMonth	月中的天，从 1 到 31
hour	从 0 到 23 表示的时
minute	从 0 到 59 表示的分
second	从 0 到 59 表示的秒

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class HelloWorld {

    public static void main(String[] args) {

        //使用 now 方法获得 LocalDate 对象
        LocalDate date1 = LocalDate.now();
        System.out.println("date1 = " + date1);

        //使用 of 方法获得 LocalDate 对象 2018 - 08 - 18
        LocalDate date2 = LocalDate.of(2018, 8, 18);
        System.out.println("date2 = " + date2);

        //使用 now 方法获得 LocalTime 对象
        LocalTime time1 = LocalTime.now();
        System.out.println("time1 = " + time1);

        //使用 of 方法获得 LocalTime 对象 08:58:18
```

```
        LocalTime time2 = LocalTime.of(8, 58, 18);
        System.out.println("time2 = " + time2);

        //使用 now 方法获得 LocalDateTime 对象
        LocalDateTime dateTime1 = LocalDateTime.now();
        System.out.println("dateTime1 = " + dateTime1);

        //使用 of 方法获得 LocalDateTime 对象 2018-08-18T08:58:18
        LocalDateTime dateTime2 = LocalDateTime.of(2018, 8, 18, 8, 58, 18);
        System.out.println("dateTime2 = " + dateTime2);
    }

}
```

运行结果如下：

```
date1 = 2017-06-04
date2 = 2018-08-18
time1 = 17:41:15.073
time2 = 08:58:18
dateTime1 = 2017-06-04T17:41:15.073
dateTime2 = 2018-08-18T08:58:18
```

从运行结果可见，日期时间输出都是本地格式。上述代码比较简单，这里不再赘述。

14.6.2 日期格式化和解析

Java 8 提供的日期格式化类是 java.time.format.DateTimeFormatter，DateTimeFormatter 中本身没有提供日期格式化和日期解析方法，这些方法还是由 LocalDate、LocalTime 和 LocalDateTime 提供的。

1. 日期格式化

日期格式化方法是 format，这 3 个类每一个都有 String format(DateTimeFormatter formatter)，参数 formatter 是 DateTimeFormatter 类型。

2. 日期解析

日期解析方法是 parse，这 3 个类每一个都有两个版本的 parse 方法，具体说明如下。

- static LocalDateTime parse(CharSequence text)：使用默认格式，从一个文本字符串获取一个 LocalDateTime 实例，如 2007-12-03T10:15:30。
- static LocalDateTime parse(CharSequence text,DateTimeFormatter formatter)：使用指定格式化，从文本字符串获取 LocalDateTime 实例。
- static LocalDate parse(CharSequence text)：使用默认格式，从一个文本字符串获取一个 LocalDate 实例，如 2007-12-03。
- static LocalDate parse(CharSequence text,DateTimeFormatter formatter)：使用指定格式化，从文本字符串获取 LocalDate 实例。
- static LocalTime parse(CharSequence text)：使用默认格式，从一个文本字符串获取一个 LocalTime 实例。

- static LocalTime parse(CharSequence text,DateTimeFormatter formatter):使用指定格式化,从文本字符串获取 LocalTime 实例。

示例代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public class HelloWorld {

    public static void main(String[] args) {

        //创建 LocalDateTime 对象
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("dateTime 格式化之前:" + dateTime);

        //设置格式化类
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String text = dateTime.format(formatter);
        System.out.println("dateTime 格式化之后:" + text);

        //格式化字符串"2018-08-18 08:58:18",返回 LocalDateTime 对象
        LocalDateTime parsedDateTime = LocalDateTime.parse("2018-08-18 08:58:18", formatter);
        System.out.println("LocalDateTime 解析之后:" + parsedDateTime);

        //创建 LocalDate 对象
        LocalDate date = LocalDate.now();
        System.out.println("date 格式化之前:" + date);

        //重新设置格式化类
        formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
        text = date.format(formatter);
        System.out.println("date 格式化之后:" + text);

        //格式化字符串"2018-08-18",返回 LocalDate 对象
        LocalDate parsedDate = LocalDate.parse("2018-08-18", formatter);
        System.out.println("LocalDate 解析之后:" + parsedDate);

        //创建 LocalTime 对象
        LocalTime time = LocalTime.now();
        System.out.println("time 格式化之前:" + time);

        //重新设置格式化类
        formatter = DateTimeFormatter.ofPattern("HH:mm:ss");                ①
        text = time.format(formatter);
        System.out.println("time 格式化之后:" + text);

        //格式化字符串"08:58:18",返回 LocalTime 对象
        LocalTime parsedTime = LocalTime.parse("08:58:18", formatter);        ②
        System.out.println("LocalTime 解析之后:" + parsedTime);
    }
}
```

运行结果如下：

```
dateTime 格式化之前:2017 - 06 - 04T18:22:25.874
dateTime 格式化之后:2017 - 06 - 04 18:22:25
LocalDateTime 解析之后:2018 - 08 - 18T08:58:18
date 格式化之前:2017 - 06 - 04
date 格式化之后:2017 - 06 - 04
LocalDate 解析之后:2018 - 08 - 18
time 格式化之前:18:22:25.891
time 格式化之后:18:22:25
LocalTime 解析之后:08:58:18
```

上述代码中格式化类 DateTimeFormatter 对象是通过 ofPattern(String pattern)获得，其中 pattern 是日期和时间格式，具体说明参考表 14-2。

注意 解析时间日期字符串时需要注意两方面的问题：第一，要解析的字符串格式一定要与格式模式匹配，假设将代码第②行的时间字符串"08:58:18"改为"08 58 18"，那么程序运行会抛出异常 DateTimeParseException；第二，要解析的字符串格式一定是有效的时间或日期，假设将代码第②行的时间字符串"08:58:18"改为"08:58:68"，那么程序运行也会抛出异常 DateTimeParse Exception。

本章小结

通过对本章的学习，可以学习到 Object 类、包装类、Math 类、BigInteger 类和 BigDecimal 类，以及旧版本日期时间类和 Java 8 之后的日期时间类。

14.7 同步练习

1. 有如下列代码：

```
public class Sample {
    long length;

    public Sample(long l) {
        length = l;
    }

    @Override
    public boolean equals(Object obj) {
        if (this.length == ((Sample) obj).length) {
            return true;
        }
        return false;
    }

    public static void main(String arg[]) {
        Sample s1, s2, s3;
```

```
        s1 = new Sample(21L);
        s2 = new Sample(21L);
        s3 = s2;
    }
}
```

在 main()方法中下列哪些表达式返回值为 true？（　　）

A. s1 == s2；　　　　B. s2 == s3；

C. s2.equals(s3)；　　　　D. s1.equals(s2)；

2. 下列哪些选择属于数值包装类？（　　）

A. Byte　　B. Short　　C. Integer　　D. Character

3. 判断对错。

（1）对货币等大数值数据进行计算时，int、long、float 和 double 等基本数据类型已经在精度方面不能满足需求了。为此 Java 提供了两个大数值类：BigInteger 和 BigDecimal，这两个类都继承自 Number 抽象类。（　　）

（2）java.sql.Date 是一个普通的日期类。（　　）

第 15 章 内　部　类

CHAPTER 15

Java 中还有一种内部类技术，简单地说内部类就是在一个类的内部又定义一个类。内部类看起来很简单，但是当你深入其中，会发现它是极其复杂的。事实上，Java 应用程序开发过程中使用内部类的地方不是很多，一般在图形用户界面开发中用于事件处理。

提示　内部类技术虽然使程序结构变得紧凑，但是却在一定程度上破坏了 Java 面向对象的思想。

15.1　内部类简介

Java 语言中允许在一个类（或方法、代码块）的内部定义另一个类，后者称为内部类（inner classes），也称为嵌套类（nested classes），封装它的类称为外部类。内部类与外部类之间存在逻辑上的隶属关系，内部类一般只在封装它的外部类或在代码块中使用。

15.1.1　内部类的作用

内部类的作用如下。

（1）封装。将不想公开的实现细节封装到一个内部类中，内部类可以声明为私有的，只能在所在外部类中访问。

（2）提供命名空间。静态内部类和外部类能够提供有别于包的命名空间。

（3）便于访问外部类成员。内部类能够很方便地访问所在外部类的成员，包括私有成员也能访问。

15.1.2　内部类的分类

内部类的分类如图 15-1 所示。按照内部类在定义的时候是否给它一个类名，可以分为有名内部类和匿名内部类。有名内部类按照作用域不同又可以分为局部内部类和成员内部类，成员内部类又分为实例成员内部类和静态成员内部类。

15.2　成员内部类

成员内部类类似于外部类的成员变量，是在外边类的内部，且方法体和代码块之外定义的内部类。

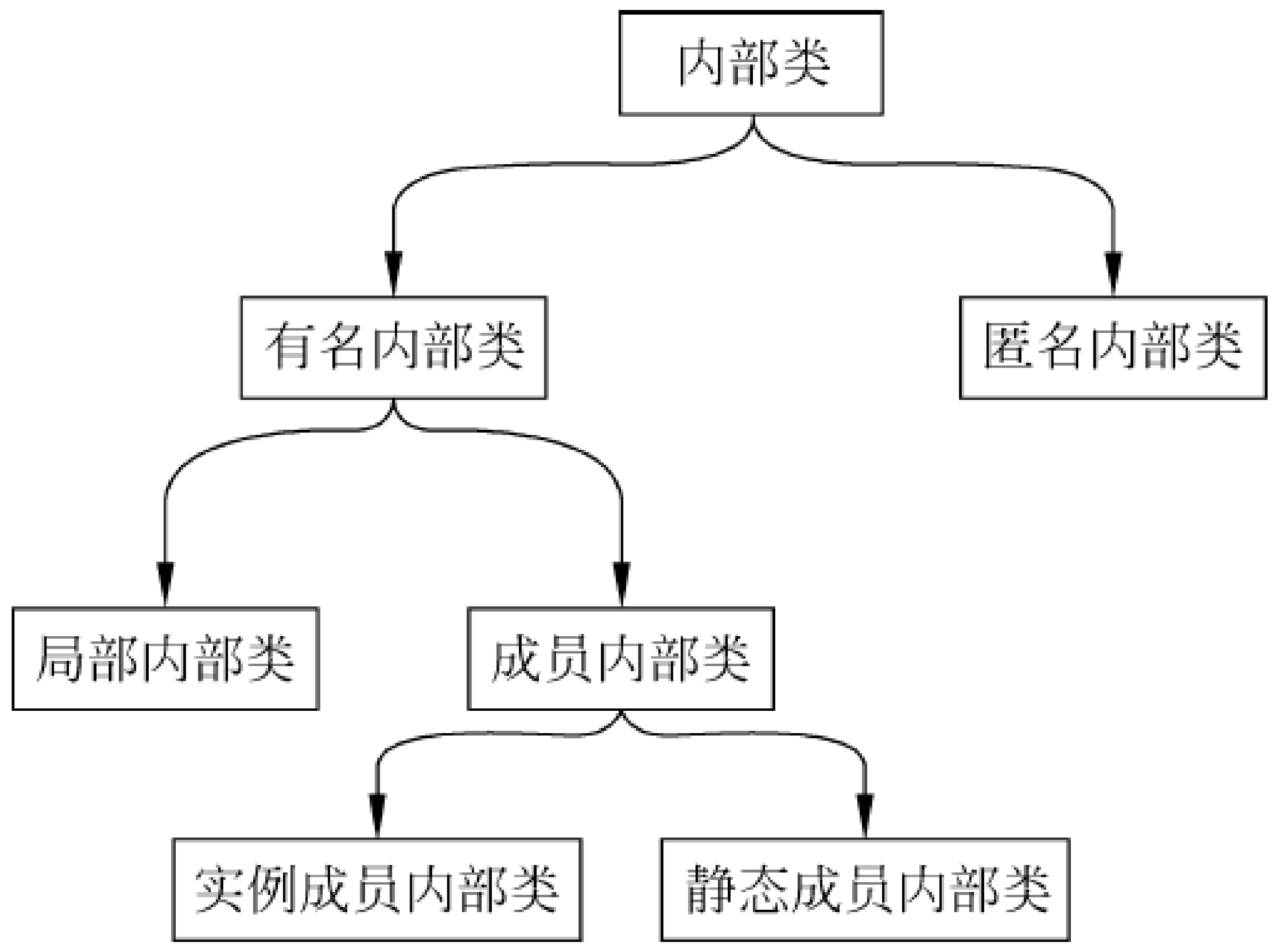


图 15-1　内部类的分类

15.2.1　实例成员内部类

实例成员内部类与实例变量类似，可以声明为公有级别、私有级别、默认级别或保护级别，即 4 种访问级别都可以，而外部类只能声明为公有或默认级别。

实例成员内部类示例代码如下：

```
//Outer.java 文件
package com.a51work6;

//外部类
public class Outer {

    //外部类成员变量
    private int x = 10;

    //外部类方法
    private void print() {
        System.out.println("调用外部方法…");
    }

    //测试调用内部类
    public void test() {
        Inner inner = new Inner();
        inner.display();
    }

    //内部类
    class Inner {                                                          ①

        //内部类成员变量
        private int x = 5;                                                 ②

        //内部类方法
        void display() {                                                   ③
```

```
            //访问外部类的成员变量 x
            System.out.println("外部类成员变量 x = " + Outer.this.x);        ④
            //访问内部类的成员变量 x
            System.out.println("内部类成员变量 x = " + this.x);              ⑤
            System.out.println("内部类成员变量 x = " + x);                   ⑥

            //调用外部类的成员方法
            Outer.this.print();                                              ⑦
            print();                                                         ⑧
        }
    }
}
```

上述代码第①行声明了内部类 Inner，它的访问级别是默认，这里还可以是 public、private 和 protected。内部类 Inner 有一个成员变量 x 和一个成员方法 display()。在 display()方法中代码第④行是访问外部类的 x 成员变量，代码第⑤行和第⑥行都是访问内部类的 x 成员变量，代码第⑦行和第⑧行都是访问外部类的 print()成员方法。

提示　在内部类中，this 是引用当前内部类对象，见代码第⑤行。而要引用外部类对象则需要使用"外部类名.this"，见代码第④行。另外，内部类和外部类的成员命名在没有冲突的情况下，在引用外部类成员时可以不用加"外部类名.this"，如代码第⑧行的 print()方法只在外部类中有定义，所以可以省略 Outer.this。

测试内部 HelloWorld 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //通过外部类访问内部类
        Outer outer = new Outer();
        outer.test();                                            ①

        System.out.println("-------直接访问内部类------");
        //直接访问内部类
        Outer.Inner inner = outer.new Inner();                   ②
        inner.display();                                         ③
    }
}
```

运行结果如下：

```
外部类成员变量 x = 10
内部类成员变量 x = 5
内部类成员变量 x = 5
调用外部方法…
```

```
调用外部方法…
-------直接访问内部类------
外部类成员变量 x = 10
内部类成员变量 x = 5
内部类成员变量 x = 5
调用外部方法…
调用外部方法…
```

通常情况下，使用实例成员内部类不是给外部类之外调用使用的，而是给外部类自己使用的。但是一定要在外部类之外访问内部类，Java 语言也是支持的，见代码第②行内部类的类型表示“外部类.内部类”，实例化过程是先实例化外部类，再实例化内部类，outer 对象是外部类实例，outer.new Inner()表达式实例化内部类对象。

另外，HelloWorld 与内部类 Inner 在同一个包中，内部类 Inner 和它的方法 display()访问级别都是默认的，它们对于在同一包中的 HelloWorld 是可见的。

提示 内部类编译成功后生成的字节码文件是“外部类$内部类.class”。

15.2.2 静态成员内部类

静态成员内部类与静态变量类似，在声明的时候使用关键字 static 修饰。静态成员内部类只能访问外部类静态成员，所以静态成员内部类使用的场景不多。但可以提供有别于包的命名空间。

示例代码如下：

```
//View.java 文件
package com.a51work6;

//外部类
public class View {

    //外部类实例变量
    private int x = 20;                                         ①
    //外部类静态变量
    private static int staticX = 10;                            ②

    //静态成员内部类
    static class Button {                                       ③

        //内部类方法
        void onClick() {                                        ④
            //访问外部类的静态成员
            System.out.println(staticX);                        ⑤
            //不能访问外部类的非静态成员
            //System.out.println(x);          //编译错误        ⑥
        }
    }
}
```

上述代码第③行定义了静态成员内部类 Button，在静态成员内部类中可以访问外部类的静态成员，见代码第⑤行。但是不能访问非静态成员，见代码第⑥行，其试图访问外部类的 x 实例变量，此时会发生编译错误。

测试内部 HelloWorld 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //直接访问内部类
        View.Button button = new View.Button();
        button.onClick();
    }
}
```

从代码 View.Button button = new View.Button()可见，在声明静态成员内部类时采用"内部类.静态成员内部类"形式，实例化也是如此形式。

提示 如果不看代码或文档，View.Button 形式看起来像是 View 包中的 Button 类，事实上它是 View 类中静态成员内部类 Button。View.Button 形式客观上能够提供有别于包的命名空间，View 相关的类集中管理起来，View.Button 可以防止命名冲突。

15.3 局部内部类

局部内部类就是在方法体或代码块中定义的内部类，局部内部类的作用域仅限于方法体或代码块中。局部内部类访问级别只能是默认的，不能是公有的、私有的和保护的访问级别，即不能使用 public、private 和 protected 修饰。局部内部类也不能是静态的，即不能使用 static 修饰。局部内部类可以访问外部类所有成员。

示例代码如下：

```
//Outer.java 文件
package com.a51work6;

//外部类
public class Outer {

    //外部类成员变量
    private int value = 10;

    //外部类方法
    public void add(final int x, int y) {                ①
        //局部变量
        int z = 100;
```

```
            //定义内部类
            class Inner {                    ②
                //内部类方法
                void display() {
                    int sum = x + z + value;  ③
                    System.out.println("sum = " + sum);
                }
            }

            //Inner inner = new Inner();
            //inner.display();
            //声明匿名对象
            new Inner().display();           ④
        }
    }
```

上述代码在 add 方法中定义了局部内部类，见代码第②行，在内部类中代码第③行访问了外部类成员变量 value、方法参数 x 和方法局部变量 z，其中方法参数应该声明为 final 的，见代码第①行 x 参数是 final 的。

提示 代码第④行 new Inner().display()是实例化 Inner 对象后立即调用它的方法，没有为 Inner 对象分配一个引用变量名，这种写法称为“匿名对象”。匿名对象适合只运行一次的情况下。匿名对象写法使代码变得简洁，但是给初学者阅读代码带来了难度。

测试内部 HelloWorld 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Outer outer = new Outer();
        outer.add(100, 300);
    }
}
```

15.4 匿名内部类

匿名内部类是没有名字的内部类，本质上是没有名的局部内部类，具有局部内部类的所有特征。例如，可以访问外部类所有成员。如果匿名内部类在方法中定义，则它所访问的参数需要声明为 final 的。

下面通过示例介绍一下匿名内部类。有如下一个 View 类：

```
//View.java 文件
package com.a51work6;
```

```
//外部类
public class View {

    public void handler(OnClickListener listener) {              ①
        listener.onClick();
    }
}
```

代码第①行中的 handler 方法需要一个实现 OnClickListener 接口的参数。OnClickListener 接口代码如下：

```
//OnClickListener.java 文件
package com.a51work6;

public interface OnClickListener {
    void onClick();
}
```

接口中只有一个 onClick()方法。使用匿名内部类的示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        View v = new View();                              ①
        //方法参数是匿名内部类
        v.handler(new OnClickListener() {                 ②

            @Override
            public void onClick() {
                System.out.println("实现接口的匿名内部类…");
            }

        });

        //继承类的匿名内部类
        Figure f = new Figure() {                         ③
            @Override
            public void onDraw() {
                System.out.println("继承类的匿名内部类…");
            }
```

```
        };

        //具体类作为内部类
        Person person = new Person("Tony", 18) {                ④
            @Override
            public String toString() {
                return "匿名内部类.实现 "
                        + " Person[name = " + name
                        + ", age = " + age + "]";
            }
        };

        //打印过程自动调用 person 的 toString()方法
        System.out.println(person);

    }
}
```

在 HelloWorld 的 main 方法中，代码第①行是实例化 View 对象，代码第②行是调用它的 handler 方法，该方法需要一个实现 OnClickListener 接口的参数，new OnClickListener() {…}表达式是实际参数，它就是匿名内部类。表达式中 OnClickListener 是要实现的接口或要继承的类，new 是为匿名内部类创建对象，()是调用构造方法，{…}是类体部分。

代码第③行是在赋值时使用匿名内部类，其中 Figure 是 13.1.2 节使用过的抽象类。匿名内部类实现了它的抽象方法 onDraw()。

代码第④行也是在赋值时使用匿名内部类，其中 Person 是 12.4.2 节使用过的具体类，匿名内部类覆盖了 toString()方法。它有两个参数的构造方法，匿名内部类使用了这个构造方法。

匿名内部类通常用来实现接口或抽象类，很少覆盖具体类。

本章小结

通过对本章的学习，了解了内部类的概念，熟悉了内部类的划分以及如何编写内部类。

15.5 同步练习

1. 实例成员内部类可以声明为(　　)。

   A. 公有级别　　　　B. 私有级别

   C. 默认级别　　　　D. 保护级别

2. 判断对错。

(1) 一个内部类可以声明为 static 的。(　　)

(2) 一个内部类可以继承其他类。(　　)

3．有如下代码：

```
class Outer {
    static class Inner {
    }
}
```

下列选项中哪些是正确的？（　　）

A．new Outer.Inner()；

B．new Outer().new Inner()；

C．new Inner()；

D．new Outer()；

第16章 Lambda 表达式

CHAPTER 16

Java 8 之后推出的 Lambda 表达式开启了 Java 语言支持函数式编程[①](functional programming)的新时代。Lambda 表达式也称为闭包(closure),现在很多语言都支持 Lambda 表达式,如 C++、C#、Swift、Objective-C 和 JavaScript 等。为什么 Lambda 表达式这么受欢迎? 这是因为 Lambda 表达式是实现支持函数式编程技术的基础。

提示 函数式编程与面向对象编程有很大的差别,函数式编程将程序代码看作数学中的函数,函数本身作为另一个函数的参数或返回值,即高阶函数。而面向对象编程是按照真实世界客观事物的自然规律进行分析,客观世界中存在什么样的实体,构建的软件系统就存在什么样的实体。即便 Java 8 之后提供了对函数式编程的支持,但是 Java 还是以面向对象为主的语言,函数式编程只是对 Java 语言的补充。

16.1 Lambda 表达式简介

与其他语言相比,Java 语言的 Lambda 表达式有着明显的区别。本节介绍 Java 语言中 Lambda 表达式的概念和具体实现方法。

16.1.1 从一个示例开始

为了理解 Lambda 表达式的概念,下面先从一个示例开始。

假设有这样的一个需求:设计一个通用方法,能够实现两个数值的加法和减法运算。Java 中方法不能单独存在,必须定义在类或接口中,考虑是一个通用方法,可以设计一个数值计算接口,其中定义该通用方法。代码如下:

```
//Calculable.java 文件
package com.a51work6;

//可计算接口
public interface Calculable {
```

① 函数式编程是一种编程范式,它将计算机运算视为函数的计算。函数编程语言最重要的基础是 λ 演算(lambda calculus)。而且 λ 演算的函数可以接受函数当作输入(参数)和输出(返回值)。和指令式编程相比,函数式编程强调函数的计算比指令的执行重要。与过程化编程相比,函数式编程里,函数的计算可随时调用。——引自于百度百科 http://baike.baidu.com/item/函数式编程。

```
    //计算两个 int 数值
    int calculateInt(int a, int b);
}
```

Calculable 接口只有一个方法 calculateInt,参数是两个 int 类型,返回值也是 int 类型。通用方法如下:

```
//HelloWorld.java 文件
...
/**
 * 通过操作符进行计算
 * @param opr 操作符
 * @return 实现 Calculable 接口对象
 */
public static Calculable calculate(char opr) {

  Calculable result;

  if (opr == '+') {
    //匿名内部类实现 Calculable 接口
    result = new Calculable() {                    ①
        //实现加法运算
        @Override
        public int calculateInt(int a, int b) {    ②
            return a + b;
        }
    };
} else {
    //匿名内部类实现 Calculable 接口
    result = new Calculable() {                    ③
        //实现减法运算
        @Override
        public int calculateInt(int a, int b) {    ④
            return a - b;
        }
    };
}

return result;
}
```

通用方法 calculate 中的参数 opr 是运算符,返回值是实现 Calculable 接口对象。代码第①行和第③行都采用匿名内部类实现 Calculable 接口。代码第②行实现加法运算。代码第④行实现减法运算。

调用通用方法代码如下:

```
//HelloWorld.java 文件
...
public static void main(String[] args) {
```

```
int n1 = 10;
int n2 = 5;

//实现加法计算 Calculable 对象
Calculable f1 = calculate('+');                    ①
//实现减法计算 Calculable 对象
Calculable f2 = calculate('-');                    ②

//调用 calculateInt 方法进行加法计算
System.out.printf("%d + %d = %d\n", n1, n2,
        f1.calculateInt(n1, n2));                  ③
//调用 calculateInt 方法进行减法计算
System.out.printf("%d - %d = %d\n", n1, n2,
        f2.calculateInt(n1, n2));                  ④
}
```

代码第①行中的 f1 是实现加法计算 Calculable 对象，代码第②行中的 f2 是实现减法计算 Calculable 对象。代码第③行和第④行才进行方法调用。

16.1.2 Lambda 表达式实现

16.1.1 节通过匿名内部类实现通用方法 calculate 代码很臃肿，Java 8 采用 Lambda 表达式可以替代匿名内部类。修改之后的通用方法 calculate 代码如下：

```
//HelloWorld.java 文件
...
/**
 * 通过操作符进行计算
 * @param opr 操作符
 * @return 实现 Calculable 接口对象
 */
public static Calculable calculate(char opr) {

    Calculable result;

    if (opr == '+') {
        //Lambda 表达式实现 Calculable 接口
        result = (int a, int b) -> {                  ①
            return a + b;
        };
    } else {
        //Lambda 表达式实现 Calculable 接口
        result = (int a, int b) -> {                  ②
            return a - b;
        };
    }

    return result;
}
```

代码第①行和第②行用 Lambda 表达式替代匿名内部类，可见代码变得非常简洁。

通过以上示例的演变可以给 Lambda 表达式一个定义：Lambda 表达式是一个匿名函数(方法)代码块，可以作为表达式、方法参数和方法返回值。

Lambda 表达式标准语法形式如下：

```
(参数列表) -> {
    //Lambda 表达式体
}
```

其中，Lambda 表达式参数列表与接口中方法参数列表形式一样，Lambda 表达式体实现接口方法。

18.1.3 函数式接口

Lambda 表达式实现的接口不是普通的接口，称为函数式接口，这种接口只能有一个方法。如果接口中声明多个抽象方法，那么 Lambda 表达式会发生编译错误：

```
The target type of this expression must be a functional interface
```

这说明该接口不是函数式接口，为了防止在函数式接口中声明多个抽象方法，Java 8 提供了一个声明函数式接口注解@FunctionalInterface，示例代码如下：

```
//Calculable.java 文件
package com.a51work6;

//可计算接口
@FunctionalInterface
public interface Calculable {
    //计算两个 int 数值
    int calculateInt(int a, int b);
}
```

在接口之前使用@FunctionalInterface 注解修饰，那么试图增加一个抽象方法时会发生编译错误，但可以添加默认方法和静态方法。

提示 Lambda 表达式是一个匿名方法代码，Java 中的方法必须声明在类或接口中，Lambda 表达式所实现的匿名方法是在函数式接口中声明的。

16.2 Lambda 表达式的简化形式

使用 Lambda 表达式是为了简化程序代码，Lambda 表达式本身也提供了多种简化形式，这些简化形式虽然简化了代码，但客观上使得代码可读性变差。本节介绍 Lambda 表达式的几种简化形式。

16.2.1 省略参数类型

Lambda 表达式可以根据上下文代码环境推断出参数类型。calculate 方法中 Lambda

表达式能推断出参数 a 和 b 是 int 类型，简化形式如下：

```
public static Calculable calculate(char opr) {

    Calculable result;

    if (opr == '+') {
        //Lambda 表达式实现 Calculable 接口
        result = (a, b) -> {                    ①
            return a + b;
        };
    } else {
        //Lambda 表达式实现 Calculable 接口
        result = (a, b) -> {                    ②
            return a - b;
        };
    }

    return result;
}
```

上述代码第①行和第②行的 Lambda 表达式是上一节示例的简化写法，其中 a 和 b 是参数。

16.2.2 省略参数小括号

当 Lambda 表达式中的参数只有一个时，可以省略参数小括号。修改 Calculable 接口，代码如下：

```
//Calculable.java 文件
package com.a51work6;

//可计算接口
@FunctionalInterface
public interface Calculable {
    //计算一个 int 数值
    int calculateInt(int a);
}
```

其中 calculateInt 方法只有一个 int 类型参数，返回值也是 int 类型。调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int n1 = 10;
```

```
        //实现二次方计算 Calculable 对象
        Calculable f1 = calculate(2);
        //实现三次方计算 Calculable 对象
        Calculable f2 = calculate(3);

        //调用 calculateInt 方法进行加法计算
        System.out.printf("%d 二次方 = %d \n", n1, f1.calculateInt(n1));
        //调用 calculateInt 方法进行减法计算
        System.out.printf("%d 三次方 = %d \n", n1, f2.calculateInt(n1));
    }

    /**
     * 通过幂计算
     * @param power 幂
     * @return 实现 Calculable 接口对象
     */
    public static Calculable calculate(int power) {

        Calculable result;

        if (power == 2) {
            //Lambda 表达式实现 Calculable 接口
            result = (int a) -> {           //标准形式              ①
                return a * a;
            };
        } else {
            //Lambda 表达式实现 Calculable 接口
            result = a -> {                //省略形式              ②
                return a * a * a;
            };
        }
        return result;
    }
}
```

上述代码第①行和第②行都是实现 Calculable 接口的 Lambda 表达式。代码第①行是标准形式，没有任何的简化。代码第②行省略了参数类型和小括号。

16.2.3　省略 return 和大括号

如果 Lambda 表达式体中只有一条语句，那么可以省略 return 和大括号。代码如下：

```
public static Calculable calculate(int power) {
    Calculable result;

    if (power == 2) {
        //Lambda 表达式实现 Calculable 接口
        result = (int a) -> {              //标准形式
            return a * a;
```

```
            };
        } else {
            //Lambda 表达式实现 Calculable 接口
            result = a -> a * a * a;        //省略形式       ①
        }
        return result;
    }
```

上述代码第①行省略了 return 和大括号，这是最简化形式的 Lambda 表达式。代码太简洁，对于初学者而言很难理解这个表达式。

16.3 作为参数使用 Lambda 表达式

Lambda 表达式常见的一种用途是作为参数传递给方法。这需要声明参数的类型为函数式接口类型。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int n1 = 10;
        int n2 = 5;

        //打印计算结果加法计算结果
        display((a, b) -> {
            return a + b;
        }, n1, n2);                                                  ①

        //打印计算结果减法计算结果
        display((a, b) -> a - b, n1, n2);                            ②

    }

    /**
     * 打印计算结果
     *
     * @param calc Lambda 表达式
     * @param n1 操作数 1
     * @param n2 操作数 2
     */
    public static void display(Calculable calc, int n1, int n2) {    ③
        System.out.println(calc.calculateInt(n1, n2));
    }

}
```

上述代码第③行定义 display 打印计算结果方法，其中参数 calc 类型是 Calculable，这个参数既可以接收实现 Calculable 接口的对象，也可以接收 Lambda 表达式，因为 Calculable 是函数式接口。

代码第①行和第②行两次调用 display 方法，它们的第一个参数都是 Lambda 表达式。

16.4　访问变量

Lambda 表达式可以访问所在外层作用域内定义的变量，包括成员变量和局部变量。

16.4.1　访问成员变量

成员变量包括实例成员变量和静态成员变量。在 Lambda 表达式中可以访问这些成员变量，此时的 Lambda 表达式与普通方法一样，可以读取成员变量，也可以修改成员变量。

示例代码如下：

```
//LambdaDemo.java 文件
package com.a51work6;

public class LambdaDemo {
    //实例成员变量
    private int value = 10;
    //静态成员变量
    private static int staticValue = 5;

    //静态方法,进行加法运算
    public static Calculable add() {                          ①

        Calculable result = (int a, int b) -> {               ②
            //访问静态成员变量,不能访问实例成员变量
            staticValue++;
            int c = a + b + staticValue; // this.value;
            return c;
        };

        return result;
    }

    //实例方法,进行减法运算
    public Calculable sub() {                                 ③

        Calculable result = (int a, int b) -> {               ④
            //访问静态成员变量和实例成员变量
            staticValue++;
            this.value++;
            int c = a - b - staticValue - this.value;
            return c;
        };
        return result;
    }
}
```

LambdaDemo类中声明一个实例成员变量value和一个静态成员变量staticValue。此外,还声明了静态方法add(见代码第①行)和实例方法sub(见代码第③行)。add方法是静态方法,静态方法中不能访问实例成员变量,所以代码第②行的Lambda表达式中不能访问实例成员变量,也不能访问实例成员方法。sub方法是实例方法,实例方法中能够访问静态成员变量和实例成员变量,所以代码第④行的Lambda表达式中可以访问这些变量,当然实例方法和静态方法也可以访问,当访问实例成员变量或实例方法时可以使用this,在不与局部变量发生冲突的情况下可以省略this。

16.4.2 捕获局部变量

对于成员变量的访问,Lambda表达式与普通方法没有区别,但是在访问外层局部变量时会发生"捕获变量"情况。在Lambda表达式中捕获变量时,会将变量当成final的,在Lambda表达式中不能修改那些捕获的变量。

示例代码如下:

```
//LambdaDemo.java 文件
package com.a51work6;

public class LambdaDemo {
    //实例成员变量
    private int value = 10;
    //静态成员变量
    private static int staticValue = 5;

    //静态方法,进行加法运算
    public static Calculable add() {
        //局部变量
        int localValue = 20;                                    ①

        Calculable result = (int a, int b) -> {
            //localValue++;           //编译错误                ②
            int c = a + b + localValue;                         ③
            return c;
        };
        return result;
    }

    //实例方法,进行减法运算
    public Calculable sub() {
        //final 局部变量
        final int localValue = 20;                              ④

        Calculable result = (int a, int b) -> {
            int c = a - b - staticValue - this.value;           ⑤
            //localValue = c;         //编译错误                ⑥
            return c;
        };
        return result;
    }
}
```

上述代码第①行和第④行都声明一个局部变量 localValue，Lambda 表达式中捕获这个变量，见代码第③行和第⑤行。不管这个变量是否显式地使用 final 修饰，它都不能在 Lambda 表达式中修改变量，所以代码第②行和第⑥行如果去掉注释，则会发生编译错误。

16.5　方法引用

Java 8 之后增加了双冒号(::)运算符，该运算符用于"方法引用"，注意不是调用方法。"方法引用"虽然没有直接使用 Lambda 表达式，但也与 Lambda 表达式有关，与函数式接口有关。

方法引用分为静态方法的方法引用和实例方法的方法引用。它们的语法形式如下：

```
类型名::静态方法                    //静态方法的方法引用
实例名::实例方法                    //实例方法的方法引用
```

注意　**被引用方法的参数列表和返回值类型必须与函数式接口方法参数列表和方法返回值类型一致。**

示例代码如下：

```
//LambdaDemo.java 文件
package com.a51work6;

public class LambdaDemo {

    //静态方法,进行加法运算
    //参数列表要与函数式接口方法 calculateInt(int a, int b)兼容
    public static int add(int a, int b) {
        return a + b;
    }

    //实例方法,进行减法运算
    //参数列表要与函数式接口方法 calculateInt(int a, int b)兼容
    public int sub(int a, int b) {
        return a - b;
    }
}
```

LambdaDemo 类中提供了一个静态方法 add 和一个实例方法 sub。这两个方法必须与函数式接口方法参数列表一致，方法返回值类型也要保持一致。

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
```

```
        int n1 = 10;
        int n2 = 5;

        //打印计算结果加法计算结果
        display(LambdaDemo::add, n1, n2);                                ①

        LambdaDemo d = new LambdaDemo();
        //打印计算结果减法计算结果
        display(d::sub, n1, n2);                                         ②

    }

    /**
     * 打印计算结果
     * @param calc Lambda 表达式
     * @param n1 操作数 1
     * @param n2 操作数 2
     */
    public static void display(Calculable calc, int n1, int n2) {        ③
        System.out.println(calc.calculateInt(n1, n2));
    }
}
```

代码第③行声明 display 方法，第一个参数 calc 是 Calculable 类型，它可以接收 3 种对象：Calculable 实现对象、Lambda 表达式和方法引用。代码第①行中第一个实际参数 LambdaDemo::add 是静态方法的方法引用。代码第②行中第一个实际参数 d::sub 是实例方法的方法引用，d 是 LambdaDemo 实例。

提示 在代码第①行的 LambdaDemo::add 和第②行的 d::sub 中，Lambda 是方法引用，此时并没有调用方法，只是将引用传递给 display 方法，在 display 方法中才真正调用方法。

本章小结

本章介绍了 Lambda 表达式，读者需要了解为什么使用 Lambda 表达式，Lambda 表达式的优点是什么；掌握 Lambda 表达式标准语法，了解 Lambda 表达式的几个简写方式；熟悉 Lambda 表达式作为参数使用的场景，了解方法引用。

16.6 同步练习

1. 判断对错。

(1) Lambda 表达式实现的接口不是普通的接口，称为函数式接口，这种接口只能有一个方法。(　　)

（2）双冒号(::)运算符用于“方法调用”。（　　）

2. 下列选项中（　　）是标准 Lambda 表达式的定义。

A.

```
(参数列表) -> {
    //Lambda 表达式体
}
```

B.

```
{ (参数列表) -> 返回值类型
    //Lambda 表达式体
}
```

C.

```
(参数列表) -> 返回值类型 {
    //Lambda 表达式体
}
```

D.

```
(参数列表) {
    //Lambda 表达式体
}
```

3. 有如下定义接口语句：

```
@FunctionalInterface
interface Calculable {
    //计算两个 int 数值
    int calculateInt(int a, int b);
}
```

下列选项中（　　）是正确的 Lambda 表达式。

A. Calculable result1 = (a) -> { return a * a; };

B. Calculable result2 = a -> { return a * a; };

C. Calculable result3 = a -> a * a ;

D. Calculable result3 = a -> {a * a};

第 17 章 异常处理

CHAPTER 17

很多事件并非总是按照人们自己的设计意图顺利发展的，而是会出现这样那样的异常情况。例如，你计划周末郊游，你的计划会安排得满满的，计划可能是这样的：从家里出发→到达目的地→游泳→烧烤→回家。但天有不测风云，当你准备烧烤时天降大雨，你只能终止郊游提前回家。“天降大雨”是一种异常情况，你的计划应该考虑到这种情况，并且应该有处理这种异常的预案。

为增强程序的健壮性，计算机程序的编写也需要考虑处理异常情况，Java 语言提供了异常处理功能。本章介绍 Java 异常处理机制。

17.1 从一个问题开始

为了学习 Java 异常处理机制，首先看看下面的程序：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

        public static void main(String[] args) {
            int a = 0;
            System.out.println(5 / a);
        }
}
```

这个程序没有编译错误，但会发生如下的运行时错误：

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.a51work6.HelloWorld.main(HelloWorld.java:9)
```

在数学上除数不能为 0，所以程序运行时表达式(5/a)会抛出 ArithmeticException 异常。ArithmeticException 是数学计算异常，凡是发生数学计算错误都会抛出该异常。

程序运行过程中难免会发生异常。发生异常并不可怕，程序员应该考虑到有可能发生这些异常，编程时应该捕获并处理异常，不能让程序发生终止，这就是健壮的程序。

17.2 异常类继承层次

异常封装成为类 Exception，此外，还有 Throwable 和 Error 类。异常类继承层次如图 17-1 所示。

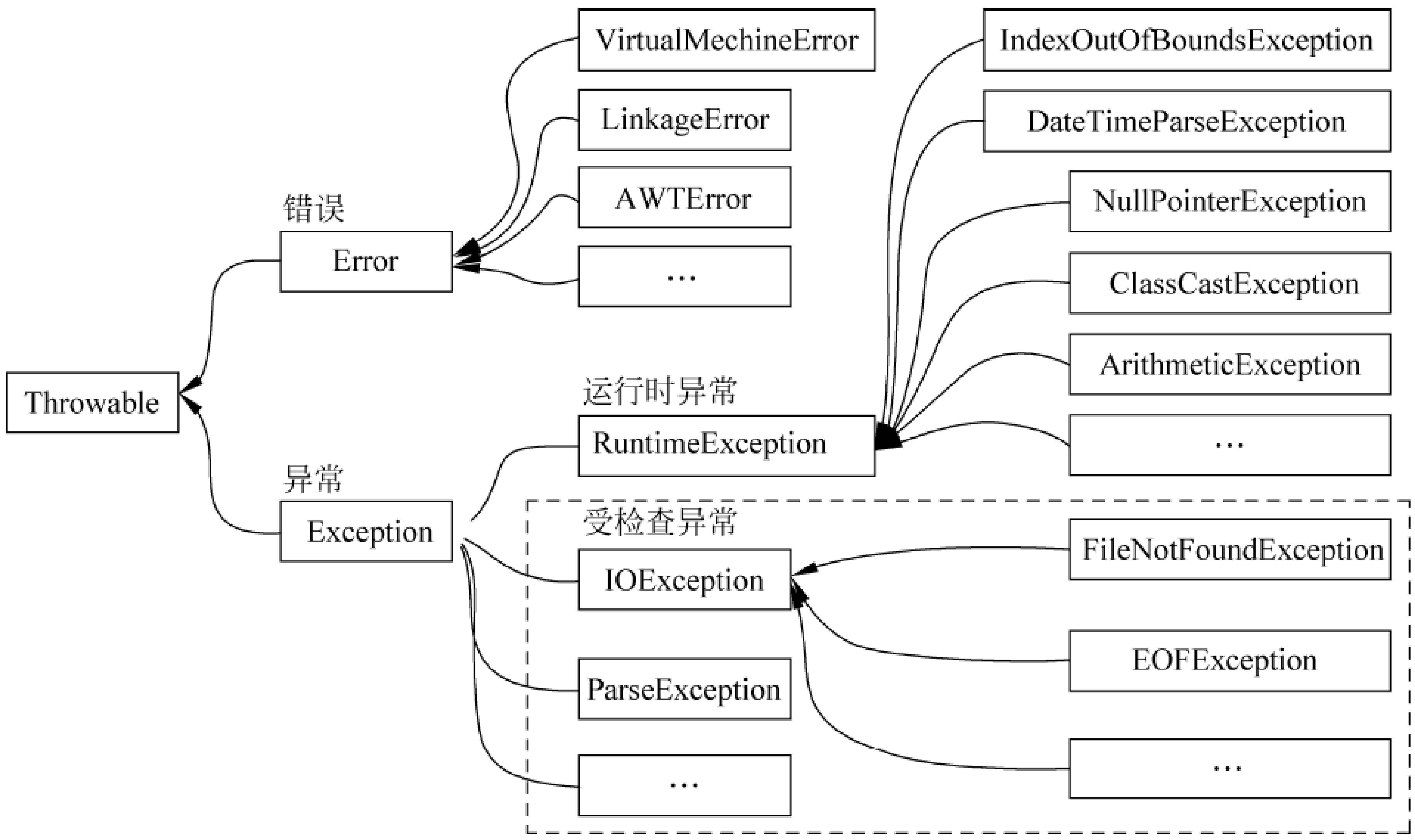


图 17-1 Java 异常类继承层次

17.2.1 Throwable 类

从图 17-1 中可见，所有的异常类都直接或间接地继承于 java.lang.Throwable 类。在 Throwable 类中有以下几个非常重要的方法。

- String getMessage()：获得发生异常的详细消息。
- void printStackTrace()：打印异常堆栈跟踪信息。
- String toString()：获得异常对象的描述。

提示 堆栈跟踪是方法调用过程的轨迹，它包含了程序执行过程中方法调用的顺序和所在源代码行号。

为了介绍 Throwable 类的使用，修改 17.1 节的示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int a = 0;
        int result = divide(5, a);
```

```
            System.out.printf("divide( %d,  %d)  =  %d", 5, a, result);
        }

        public static int divide(int number, int divisor) {

            try {
                return number / divisor;
            } catch (Throwable throwable) {                                    ①

                System.out.println("getMessage() : " + throwable.getMessage());  ②

                System.out.println("toString() : " + throwable.toString());      ③

                System.out.println("printStackTrace()输出信息如下:");
                throwable.printStackTrace();                                   ④
            }

            return 0;
        }
    }
```

运行结果如下：

```
getMessage() : / by zero
toString() : java.lang.ArithmeticException: / by zero
printStackTrace()输出信息如下:
java.lang.ArithmeticException: / by zero
    at com.a51work6.HelloWorld.divide(HelloWorld.java:17)
    at com.a51work6.HelloWorld.main(HelloWorld.java:10)
divide(5, 0)  =  0
```

将可以发生异常的语句 number/divisor 放到 try-catch 代码块中，称为捕获异常，有关捕获异常的知识将在下一节详细介绍。在 catch 中有一个 Throwable 对象 throwable，throwable 对象是系统在程序发生异常时创建的，通过 throwable 对象可以调用 Throwable 中定义的方法。

代码第②行是调用 getMessage()方法获得异常消息，输出结果是“/ by zero”。代码第③行是调用 toString()方法获得异常对象的描述，输出结果是 java.lang.ArithmeticException:/by zero。代码第④行是调用 printStackTrace()方法打印异常堆栈跟踪信息。

提示 堆栈跟踪信息从下往上，是方法调用的顺序。首先 JVM 调用的是 com.a51work6.HelloWorld 类的 main 方法，接着在 HelloWorld.java 源代码第 10 行调用 com.a51work6.HelloWorld 类的 divide 方法，在 HelloWorld.java 源代码第 17 行发生了异常，最后输出的是异常信息。

17.2.2 Error 和 Exception

从图 17-1 中可见，Throwable 有两个直接子类：Error 和 Exception。

1. Error

Error 是程序无法恢复的严重错误，程序员对其根本无能为力，只能让程序终止，如 JVM 内部错误、内存溢出和资源耗尽等严重情况。

2. Exception

Exception 是程序可以恢复的异常，它是程序员所能掌控的，如除零异常、空指针访问、网络连接中断和读取不存在的文件等。本章所讨论的异常处理就是对 Exception 及其子类的异常处理。

17.2.3 受检查异常和运行时异常

从图 17-1 中可见，Exception 类可以分为受检查异常和运行时异常。

1. 受检查异常

如图 17-1 所示，受检查异常是除 RuntimeException 以外的异常类。它们的共同特点是：编译器会检查这类异常是否进行了处理，即要么捕获（try-catch 语句），要么抛出（通过在方法后声明 throws），否则会发生编译错误。它们种类很多，如前面遇到过的日期解析异常 ParseException。

2. 运行时异常

运行时异常是继承 RuntimeException 类的直接或间接子类。运行时异常往往是程序员所犯错误导致的，健壮的程序不应该发生运行时异常。它们的共同特点是：编译器不检查这类异常是否进行了处理，也就是对于这类异常不捕获也不抛出，程序也可以编译通过。由于没有进行异常处理，一旦运行时异常发生就会导致程序的终止，这是用户不希望看到的。由于 17.2.1 节除零示例的 ArithmeticException 异常属于 RuntimeException 异常，如图 17-1 所示，可以不用加 try-catch 语句捕获异常。

对于运行时异常通常不采用抛出或捕获处理方式，而是应该提前预判，防止发生这种异常，做到未雨绸缪。例如 17.2.1 节除零示例，在进行除法运算之前应该判断除数是非零的，提前预判这种处理方式要比通过 try-catch 捕获异常友好得多。修改示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        int a = 0;
        int result = divide(5, a);
        System.out.printf("divide(%d, %d) = %d", 5, a, result);
    }

    public static int divide(int number, int divisor) {
```

```
        //判断除数 divisor 非零,防止运行时异常
        if (divisor != 0) {
            return number / divisor;
        }
        return 0;
    }

}
```

除了图 17-1 所示异常外还有很多异常,这里不能一一穷尽。随着学习的深入会介绍一些常用的异常,关于其他异常读者可以自己查阅 API 文档。

17.3 捕获异常

在学习本节内容之前,先考虑一下,在现实生活中你是如何对待领导交给你的任务的呢?当然无非是两种方法:自己有能力解决的自己处理;自己无能力解决的反馈给领导,让领导自己处理。

对待受检查异常也是如此。当前方法有能力解决,则捕获异常进行处理;当前方法没有能力解决,则抛给上层调用方法处理。如果上层调用方法也无能力解决,则继续抛给它的上层调用方法,异常就是这样向上传递直到有方法处理它。如果所有的方法都没有处理该异常,那么 JVM 会终止程序运行。

本节先介绍如何捕获异常。

17.3.1 try-catch 语句

捕获异常是通过 try-catch 语句实现的,最基本的 try-catch 语句语法如下:

```
try{
    //可能会发生异常的语句
} catch(Throwable e){
    //处理异常 e
}
```

1. try 代码块

try 代码块中应该包含执行过程中可能会发生异常的语句。一条语句是否有可能发生异常,这要看语句中调用的方法。例如,日期格式化类 DateFormat 的日期解析方法 parse(),其完整定义如下:

```
public Date parse(String source) throws ParseException
```

方法后面的 throws ParseException 说明:当调用 parse()方法时有可能产生 ParseException 异常。

提示 静态方法、实例方法和构造方法都可以声明抛出异常,凡是抛出异常的方法都可以通过 try-catch 进行捕获,当然运行时异常可以不捕获。一个方法声明抛出什么样的异常需要查阅 API 文档。

2. catch 代码块

每个 try 代码块可以伴随一个或多个 catch 代码块，用于处理 try 代码块中可能发生的多种异常。catch(Throwable e)语句中的 e 是捕获异常对象，e 必须是 Throwable 的子类，异常对象 e 的作用域在该 catch 代码块中。

下面看一个 try-catch 示例，代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class HelloWorld {

        public static void main(String[] args) {
        Date date = readDate();
        System.out.println("日期 = " + date);
    }

    //解析日期
    public static Date readDate() {                          ①

        try {
            String str = "2018-8-18"; //"201A-18-18"
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            //从字符串中解析日期
            Date date = df.parse(str);                       ②
            return date;
        } catch (ParseException e) {                         ③
            System.out.println("处理 ParseException…");
            e.printStackTrace();                             ④
        }
        return null;
    }
}
```

上述代码第①行定义了一个静态方法用来将字符串解析成日期，但并非所有的字符串都是有效的日期字符串，因此调用代码第②行的解析方法 parse()有可能发生 ParseException 异常，ParseException 是受检查异常，在本例中使用 try-catch 捕获。代码第③行的 e 就是 ParseException 对象。代码第④行的 e.printStackTrace()是打印异常堆栈跟踪信息，本例中的"2018-8-18"字符串是一个有效的日期字符串，因此不会发生异常。如果将字符串改为无效的日期字符串，如"201A-18-18"，则会打印信息如下：

```
处理 ParseException
java.text.ParseException: Unparseable date: "201A-18-18"
日期 = null
    at java.text.DateFormat.parse(Unknown Source)
    at com.a51work6.HelloWorld.readDate(HelloWorld.java:24)
    at com.a51work6.HelloWorld.main(HelloWorld.java:13)
```

提示 在捕获到异常之后，通过e.printStackTrace()语句打印异常堆栈跟踪信息，往往只是用于调试，给程序员提示信息。堆栈跟踪信息对最终用户是没有意义的，本例中如果出现异常很有可能是用户输入的日期无效，捕获到异常之后给用户弹出一个对话框，提示用户输入日期无效，请用户重新输入，用户重新输入后再重新调用上述方法。这才是捕获异常之后的正确处理方案。

17.3.2 多 catch 代码块

如果 try 代码块中有很多语句会发生异常，而且发生的异常种类又很多，那么可以在 try 后面跟有多个 catch 代码块。多 catch 代码块语法如下：

```
try{
    //可能会发生异常的语句
} catch(Throwable e){
    //处理异常 e
} catch(Throwable e){
    //处理异常 e
} catch(Throwable e){
    //处理异常 e
}
```

在多个 catch 代码块情况下，当一个 catch 代码块捕获到一个异常时，其他的 catch 代码块就不再进行匹配。

注意 当捕获的多个异常类之间存在父子关系时，捕获异常顺序与 catch 代码块的顺序有关。一般先捕获子类，后捕获父类，否则捕获不到子类。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

...

public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期 = " + date);
    }
    public static Date readDate() {

        FileInputStream readfile = null;
```

```
            InputStreamReader ir = null;
            BufferedReader in = null;
            try {
                readfile = new FileInputStream("readme.txt");              ①
                ir = new InputStreamReader(readfile);
                in = new BufferedReader(ir);
                //读取文件中的一行数据
                String str = in.readLine();                                ②
                if (str == null) {
                    return null;
                }

                DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
                Date date = df.parse(str);                                 ③
                return date;

            } catch (FileNotFoundException e) {                            ④
                System.out.println("处理 FileNotFoundException...");
                e.printStackTrace();
            } catch (IOException e) {                                      ⑤
                System.out.println("处理 IOException...");
                e.printStackTrace();
            } catch (ParseException e) {                                   ⑥
                System.out.println("处理 ParseException...");
                e.printStackTrace();
            }
            return null;
        }

    }
```

上述代码通过 Java I/O(输入输出)流技术从文件 readme.txt 中读取字符串,然后解析成为日期。由于 Java I/O 技术还没有介绍,可先不关注 I/O 技术细节,只考虑调用它们的方法会发生异常即可。

在 try 代码块中第①行代码调用 FileInputStream 构造方法可能会发生 FileNotFoundException 异常。第②行代码调用 BufferedReader 输入流的 readLine()方法可能会发生 IOException 异常。从图 17-1 中可见,FileNotFoundException 异常是 IOException 异常的子类,应该先捕获 FileNotFoundException,见代码第④行;后捕获 IOException,见代码第⑤行。

如果将 FileNotFoundException 和 IOException 捕获顺序调换,代码如下:

```
try{
    //可能会发生异常的语句
} catch (IOException e) {
    //IOException 异常处理
} catch (FileNotFoundException e) {
    //FileNotFoundException 异常处理
}
```

那么第二个 catch 代码块永远不会进入,FileNotFoundException 异常处理永远不会

执行。

由于示例代码第⑥行的 ParseException 异常与 IOException 和 FileNotFoundException 异常没有父子关系，捕获 ParseException 异常位置可以随意放置。

17.3.3 try-catch 语句嵌套

Java 提供的 try-catch 语句嵌套可以任意嵌套，修改 17.3.2 节示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;
…
public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期 = " + date);
    }

    public static Date readDate() {

        FileInputStream readfile = null;
        InputStreamReader ir = null;
        BufferedReader in = null;
        try {
            readfile = new FileInputStream("readme.txt");
            ir = new InputStreamReader(readfile);
            in = new BufferedReader(ir);

            try {                                                    ①
                String str = in.readLine();                          ②
                if (str == null) {
                    return null;
                }

                DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
                Date date = df.parse(str);                           ③
                return date;

            } catch (ParseException e) {
                System.out.println("处理 ParseException…");
                e.printStackTrace();
            }                                                        ④
        } catch (FileNotFoundException e) {                          ⑤
            System.out.println("处理 FileNotFoundException…");
            e.printStackTrace();
        } catch (IOException e) {                                    ⑥
            System.out.println("处理 IOException…");
            e.printStackTrace();
        }
        return null;
    }
}
```

上述代码第①～④行是捕获 ParseException 异常 try-catch 语句，可见这个 try-catch 语句就是嵌套在捕获 IOException 和 FileNotFoundException 异常的 try-catch 语句中。

程序执行时内层如果会发生异常，首先由内层 catch 进行捕获，如果捕获不到，则由外层 catch 捕获。例如，代码第②行的 readLine()方法可能发生 IOException 异常，该异常无法被内层 catch 捕获，最后被代码第⑥行的外层 catch 捕获。

注意 try-catch 不仅可以嵌套在 try 代码块中，还可以嵌套在 catch 代码块或 finally 代码块中，finally 代码块后面会详细介绍。try-catch 嵌套会使程序流程变得复杂，如果能用多 catch 捕获的异常，尽量不要使用 try-catch 嵌套。特别对于初学者，不要简单地使用 Eclipse 的语法提示不加区分地添加 try-catch 嵌套，要先梳理好程序的流程再考虑 try-catch 嵌套的必要性。

17.3.4 多重捕获

多 catch 代码块客观上提高了程序的健壮性，但是程序代码量大大增加。有些异常虽然种类不同，但捕获之后的处理是相同的，看如下代码：

```
try{
    //可能会发生异常的语句
} catch (FileNotFoundException e) {
    //调用方法 methodA 处理
} catch (IOException e) {
    //调用方法 methodA 处理
} catch (ParseException e) {
    //调用方法 methodA 处理
}
```

3 个不同类型的异常，要求捕获之后的处理都是调用 methodA 方法。是否可以把这些异常合并处理，Java 7 推出了多重捕获(multi-catch)技术，可以帮助解决此类问题，上述代码修改如下：

```
try{
    //可能会发生异常的语句
} catch (IOException | ParseException e) {
    //调用方法 methodA 处理
}
```

在 catch 中多重捕获异常用“|”运算符连接起来。

注意 有的读者会问为什么不写成 FileNotFoundException | IOException | ParseException 呢？这是因为 FileNotFoundException 属于 IOException 异常，IOException 异常可以捕获它的所有子类异常。

17.4 释放资源

有时在 try-catch 语句中会占用一些非 Java 资源，如打开文件、网络连接、打开数据库连接和使用数据结果集等，这些资源并非 Java 资源，不能通过 JVM 的垃圾收集器回收，需

要程序员释放。为了确保这些资源能够被释放，可以使用 finally 代码块或 Java 7 之后提供的自动资源管理(automatic resource management)技术。

17.4.1 finally 代码块

try-catch 语句后面还可以跟有一个 finally 代码块。try-catch-finally 语句语法如下：

```
try{
    //可能会生成异常语句
} catch(Throwable e1){
    //处理异常 e1
} catch(Throwable e2){
    //处理异常 e2
⋮
} catch(Throwable eN){
    //处理异常 eN
} finally{
    //释放资源
}
```

无论 try 正常结束还是 catch 异常结束都会执行 finally 代码块，如图 17-2 所示。

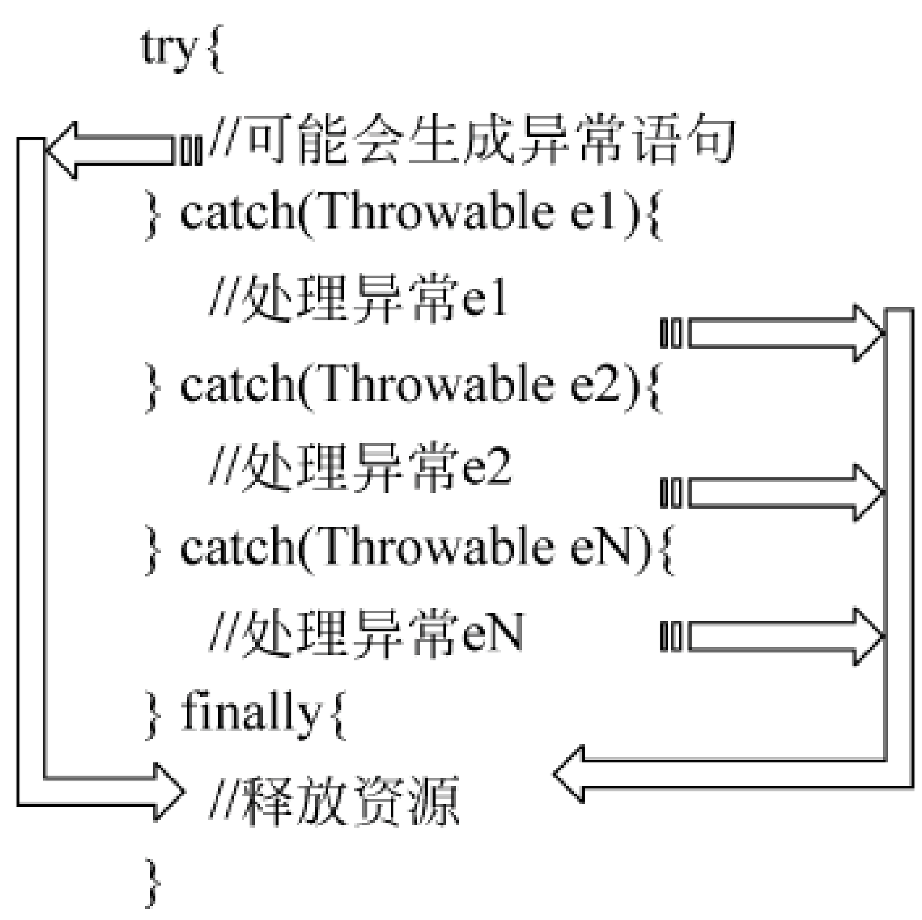


图 17-2 finally 代码块流程

使用 finally 代码块示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

…

public class HelloWorld {

    public static void main(String[ ] args) {
        Date date = readDate();
        System.out.println("读取的日期 = " + date);
    }

    public static Date readDate() {

        FileInputStream readfile = null;
```

```
        InputStreamReader ir = null;
        BufferedReader in = null;
        try {
            readfile = new FileInputStream("readme.txt");
            ir = new InputStreamReader(readfile);
            in = new BufferedReader(ir);
            //读取文件中的一行数据
            String str = in.readLine();
            if (str == null) {
                return null;
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);
            return date;

        } catch (FileNotFoundException e) {
            System.out.println("处理 FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("处理 IOException...");
            e.printStackTrace();
        } catch (ParseException e) {
            System.out.println("处理 ParseException...");
            e.printStackTrace();
        } finally {                                    ①
            try {
                if (readfile != null) {
                    readfile.close();                  ②
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                if (ir != null) {
                    ir.close();                        ③
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                if (in != null) {
                    in.close();                        ④
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }                                              ⑤

        return null;
    }
}
```

上述代码第①～⑤行是 finally 语句，在这里通过关闭流释放资源。FileInputStream、InputStreamReader 和 BufferedReader 是 3 个输入流，它们都需要关闭，见代码第②～④行通过流的 close()关闭流，但是流的 close()方法还有可能发生 IOException 异常，所以这里针对每一个 close()语句还需要进行捕获处理。

注意 为了使代码简洁，可能有的人会将 finally 代码中多个嵌套的 try-catch 语句合并，例如将上述代码改成如下形式，将 3 个有可能发生异常的 close()方法放到一个 try-catch。读者自己考虑一下这样处理是否稳妥呢？每一个 close()方法对应关闭一个资源，如果第一个 close()方法关闭时发生了异常，那么后面的两个也不会关闭，因此如下的程序代码是有缺陷的。

```
try {
    …
} catch (FileNotFoundException e) {
    …
} catch (IOException e) {
    …
} catch (ParseException e) {
    …
} finally {
    try {
        if (readfile != null) {
            readfile.close();
        }
        if (ir != null) {
            ir.close();
        }
        if (in != null) {
            in.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

17.4.2 自动资源管理

17.4.1 节使用 finally 代码块释放资源会导致程序代码大量增加，一个 finally 代码块往往比正常执行的程序还要多。在 Java 7 之后提供的自动资源管理技术可以替代 finally 代码块，优化代码结构，提高程序可读性。

自动资源管理是在 try 语句上的扩展，语法如下：

```
try (声明或初始化资源语句) {
    //可能会生成异常语句
} catch(Throwable e1){
    //处理异常 e1
} catch(Throwable e2){
    //处理异常 e2
 ⋮
} catch(Throwable eN){
    //处理异常 eN
}
```

在 try 语句后面添加一对小括号“()”，其中是声明或初始化资源语句，可以有多条语句，语句之间用分号(;)分隔。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;
…
public class HelloWorld {

    public static void main(String[] args) {
        Date date = readDate();
        System.out.println("读取的日期 = " + date);
    }

    public static Date readDate() {

        //自动资源管理
        try (FileInputStream readfile = new FileInputStream("readme.txt");          ①
                InputStreamReader ir = new InputStreamReader(readfile);             ②
                BufferedReader in = new BufferedReader(ir)) {                       ③

            //读取文件中的一行数据
            String str = in.readLine();
            if (str == null) {
                return null;
            }

            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            Date date = df.parse(str);
            return date;

        } catch (FileNotFoundException e) {
            System.out.println("处理 FileNotFoundException...");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("处理 IOException...");
            e.printStackTrace();
        } catch (ParseException e) {
            System.out.println("处理 ParseException...");
            e.printStackTrace();
        }

        return null;
    }

}
```

上述代码第①～③行是声明或初始化 3 个输入流，3 条语句放在 try 语句后面的小括号中，语句之间用分号(;)分隔，这就是自动资源管理技术。采用自动资源管理后不再需要

finally 代码块，不需要自己关闭这些资源，释放过程交给了 JVM。

注意 所有可以自动管理的资源需要实现 AutoCloseable 接口，上述代码中 3 个输入流 FileInputStream、InputStream Reader 和 BufferedReader 从 Java 7 之后实现 AutoCloseable 接口，具体哪些资源实现 AutoCloseable 接口需要查阅 API 文档。

17.5 throws 与声明方法抛出异常

在一个方法中如果能够处理异常，则需要捕获并处理。但是若方法没有能力处理该异常，捕获它没有任何意义，此时需要在方法后面声明抛出该异常，通知上层调用者该方法有可能发生异常。

方法后面声明抛出使用 throws 关键字，回顾一下 10.3.3 节成员方法语法格式如下：

```
class className {

    [public | protected | private ] [static] [final | abstract] [native] [synchronized]
        type methodName([paramList]) [throws exceptionList] {
            //方法体
    }
}
```

其中参数列表之后的[throws exceptionList]语句是声明抛出异常。方法中可能抛出的异常(除了 Error 和 RuntimeException 及其子类外)都必须通过 throws 语句列出，多个异常之间采用逗号(,)分隔。

如果将 17.3 节示例进行修改，在 readDate()方法后声明抛出异常，则代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

…
public class HelloWorld {

    public static void main(String[] args) {                          ①

        try {
            Date date = readDate();                                    ②
            System.out.println("读取的日期 = " + date);
        } catch (IOException e) {                                      ③
            System.out.println("处理 IOException…");
            e.printStackTrace();
        } catch (ParseException e) {                                   ④
            System.out.println("处理 ParseException…");
            e.printStackTrace();
        }

    }
```

```
    public static Date readDate() throws IOException, ParseException {        ⑤

        //自动资源管理
        FileInputStream readfile = new FileInputStream("readme.txt");         ⑥
        InputStreamReader ir = new InputStreamReader(readfile);
        BufferedReader in = new BufferedReader(ir);

        //读取文件中的一行数据
        String str = in.readLine();                                            ⑦
        if (str == null) {
            return null;
        }

        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        Date date = df.parse(str);                                             ⑧
        return date;
    }

}
```

注意　如果声明抛出的多个异常类之间有父子关系，可以只声明抛出父类。但在没有父子关系的情况下，最好明确声明抛出每一个异常，因为上层调用者会根据这些异常信息进行相应的处理。假如一个方法中有可能抛出 IOException 和 ParseException 两个异常，那么是声明抛出 IOException 和 ParseException，还是只声明抛出 Exception 呢？因为 Exception 是 IOException 和 ParseException 的父类，只声明抛出 Exception 从语法上是允许的，但是声明抛出 IOException 和 ParseException 更好一些。

由于 readDate()方法中代码第⑥～⑧行都有可能引发异常，在 readDate()方法内又没有捕获处理，所以需要在代码第⑤行方法后声明抛出异常。事实上有 3 个异常，即 FileNotFoundException、IOException 和 ParseException，由于 FileNotFoundException 属于 IOException 异常，所以只声明 IOException 和 ParseException 即可。

一旦 readDate()方法声明抛出了异常，那么它的调用者 main()方法也会面临同样的问题：要么捕获自己处理，要么抛给上层调用者。如果一旦发生异常，main()方法也选择抛出，那么程序运行就会终止。本例中 main()方法是捕获异常进行处理，捕获异常过程前面已经介绍过了，这里不再赘述。

17.6　自定义异常类

有些公司为了提高代码的可重用性，自己开发了一些 Java 类库或框架，其中少不了要编写一些异常类。实现自定义异常类需要继承 Exception 类或其子类，如果自定义运行时异常类，则需继承 RuntimeException 类或其子类。

实现自定义异常类示例代码如下：

```
package com.a51work6;

public class MyException extends Exception {                 ①
```

```
    public MyException() {                ②

    }

    public MyException(String message) {  ③
        super(message);
    }

}
```

上述代码实现了自定义异常类。自定义异常类一般需要提供两个构造方法：一个是代码第②行的无参数的默认构造方法，异常描述信息是空的；另一个是代码第③行的字符串参数的构造方法，message是异常描述信息，getMessage()方法可以获得这些信息。

自定义异常类就这样简单，主要是提供两个构造方法即可。

17.7 throw与显式抛出异常

Java异常相关的关键字中有两个非常相似，即throws和throw，其中throws关键字在17.5节已经介绍了，throws用于方法后声明抛出异常，而throw关键字用来人工引发异常。本节之前接触到的异常都是由于系统生成的，当异常发生时，系统会生成一个异常对象，并将其抛出。但也可以通过throw语句显式抛出异常。语法格式如下：

```
throw Throwable 或其子类的实例
```

所有Throwable或其子类的实例都可以通过throw语句抛出。

显式抛出异常的目的有很多，例如不想将某些异常传给上层调用者，可以捕获之后重新显式抛出另外一种异常给上层调用者。

修改17.4节示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;
…
public class HelloWorld {

    public static void main(String[] args) {
        try {
            Date date = readDate();
            System.out.println("读取的日期 = " + date);
        } catch (MyException e) {
            System.out.println("处理 MyException…");
            e.printStackTrace();
        }
    }
```

```
        public static Date readDate() throws MyException {
            //自动资源管理
            try (FileInputStream readfile = new FileInputStream("readme.txt");
                    InputStreamReader ir = new InputStreamReader(readfile);
                    BufferedReader in = new BufferedReader(ir)) {

                //读取文件中的一行数据
                String str = in.readLine();
                if (str == null) {
                    return null;
                }

                DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
                Date date = df.parse(str);
                return date;

            } catch (FileNotFoundException e) {             ①
                throw new MyException(e.getMessage());      ②
            } catch (IOException e) {                       ③
                throw new MyException(e.getMessage());      ④
            } catch (ParseException e) {
                System.out.println("处理 ParseException…");
                e.printStackTrace();
            }
            return null;
        }
    }
```

如果软件设计者不希望 readDate()方法中捕获的 FileNotFoundException 和 IOException 异常出现在 main()方法(上层调用者)中,那么可以在捕获到 FileNotFoundException 和 IOException 异常时,通过 throw 语句显式抛出一个异常,见代码第②行和第④行的 throw new MyException(e.getMessage())语句,MyException 是自定义的异常。

注意 throw 显式抛出的异常与系统生成并抛出的异常在处理方式上没有区别,就是两种方法:要么捕获自己处理,要么抛给上层调用者。在本例中是声明抛出,所以在 readDate()方法后面要声明抛出 MyException 异常。

本章小结

本章介绍了 Java 异常处理机制,其中包括 Java 异常类继承层次、捕获异常、释放资源、throws、throw 和自定义异常类。需要重点掌握捕获异常处理,熟悉 throws 和 throw 的区分和用法。

17.8 同步练习

1. 如果下列的方法能够正常运行,在控制台上将显示(　　)。

```
public class HelloWorld {
    public static void main(String[] args) {
        try {
            int a = 0;
            System.out.println(5 / a);
            System.out.println("Test1");
        } catch (Exception e) {
            System.out.println("Test 2");
        } finally {
            System.out.println("Test 3");
        }
        System.out.println("Test 4");
    }
}
```

A. Test 1　　B. Test 2　　C. Test 3　　D. Test 4

2. 哪个关键字可以抛出异常?(　　)

A. transient　　B. finally　　C. throw　　D. static

3. 下面程序的输出是(　　)。

```
class MyException extends Exception {}

public class HelloWorld {

    public static void main(String[] args) {
        try {
            throw new MyException();
        } catch (Exception e) {
            System.out.println("异常…");
        } finally {
            System.out.println("完成…");
        }
    }
}
```

A. 异常…　　B. 完成…　　C. 异常…完成…　　D. 无输出

4. 下面的程序是一个异常嵌套处理的例子,其运行结果是(　　)。

```
public class HelloWorld {

    public static void main(String args[]) {
        try {
            try {
```

```
                int i;
                int j = 0;
                i = 1 / j;
            } catch (Exception e) {
                System.out.print("1");
                throw e;
            } finally {
                System.out.print("2");
            }
        } catch (Exception e) {
            System.out.print("3");
        } finally {
            System.out.println("4");
        }
    }

}
```

A. 12　　B. 1234　　C. 234　　D. 1342

5. 下列代码在运行时抛出的异常是(　　)。

```
public class HelloWorld {
    public static void main(String args[]) {
        int a[] = new int[10];
        a[10] = 0;
    }
}
```

A. ArithmeticException　　B. ArrayIndexOutOfBoundsException

C. NegativeArraySizeException　　D. IllegalArgumentException

6. 请简述 error 和 exception 的区别。

第 18 章 CHAPTER 18 集　合

当你有很多书时，你会考虑买一个书柜，将其分门别类地摆放。书柜不仅使房间变得整洁，也便于以后使用书时查找。在计算机中管理对象亦是如此，当获得多个对象后，也需要一个容器将它们管理起来，这个容器就是集合。

集合本质上是基于某种数据结构的数据容器。常见的数据结构有数组(Array)、集合(Set)、队列(Queue)、链表(Linkedlist)、树(Tree)、堆(Heap)、栈(Stack)和映射(Map)等。

本章介绍 Java 中的集合。

18.1　集合简介

Java 中提供了丰富的集合接口和类，它们来自于 java.util 包。图 18-1 所示是 Java 主要的集合接口和类。从图中可见，Java 集合类型分为 Collection 和 Map，Collection 子接口有 Set、Queue 和 List 等接口。每一种集合接口描述了一种数据结构。

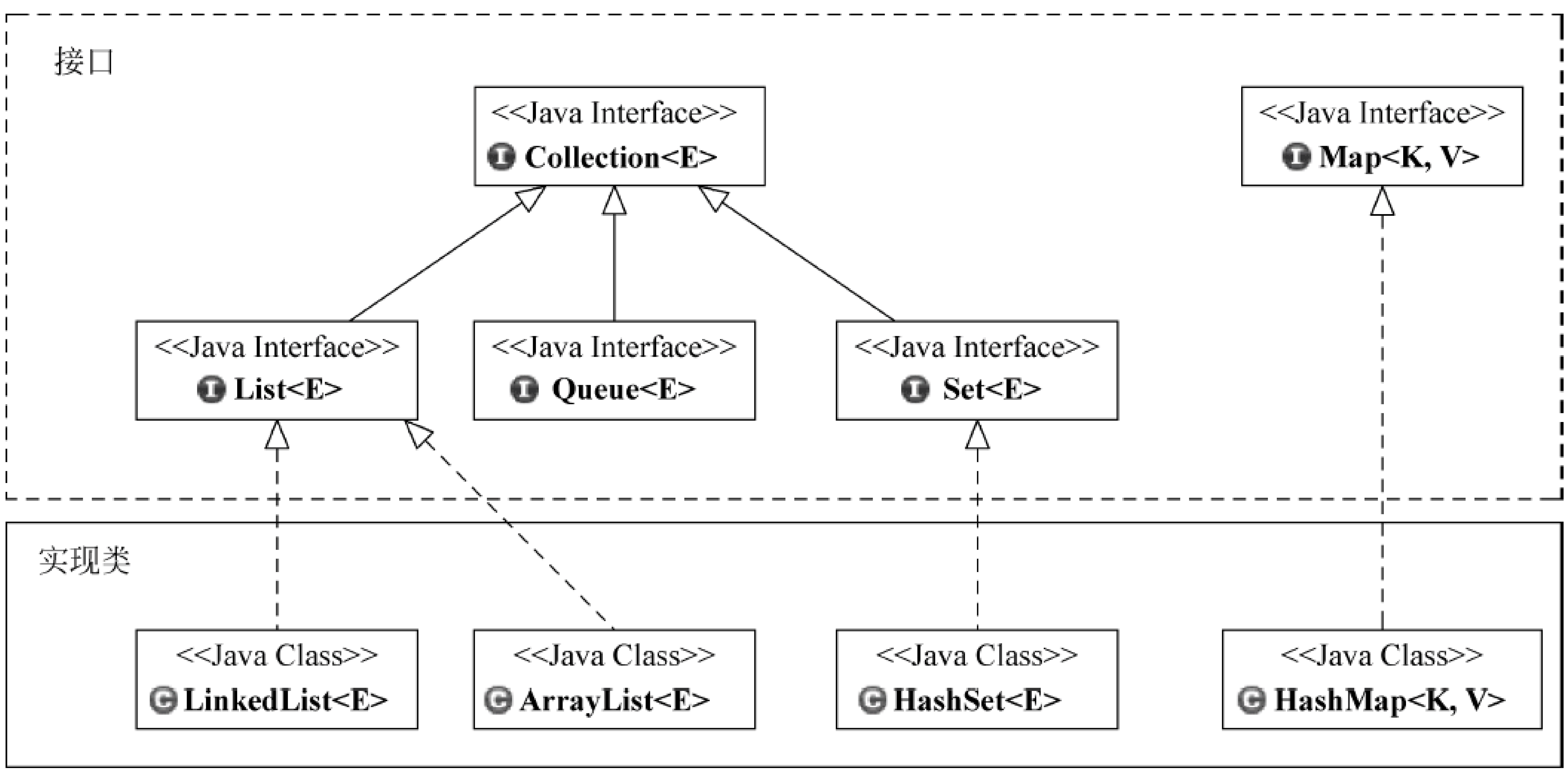


图 18-1　Java 主要集合接口和类

本章重点介绍 List、Set 和 Map 接口，因此图 18-1 中只列出了这 3 个接口的具体实现类。事实上 Queue 也有具体实现类，由于很少使用，这里不再赘述，感兴趣的读者可以自己查阅 API 文档。

提示 在 Java SE 中 List 名称的类型有两个，一个是 java.util.List，另一个是 java.awt.List。java.util.List 是一个接口，也就是本章介绍的 List 集合。而 java.awt.List 是一个类，用于图形用户界面开发，它是一个图形界面中的组件。

学习 Java 中的集合，首先从接口入手，重点掌握 List、Set 和 Map 3 个接口，熟悉这些接口中提供的方法。然后再熟悉这些接口的实现类，并了解不同实现类之间的区别。

18.2 List 集合

List 集合中的元素是有序的，可以重复出现。图 18-2 所示是一个班级集合数组，这个集合中有一些学生，这些学生是有序的，即他们被放到集合中的顺序，可以通过序号访问他们。这就像老师给进入班级的学生分配学号：第一个报到的是“张三”，老师给他分配的是 0；第二个报到的是“李四”，老师给他分配的是 1；以此类推，最后一个序号应该是“学生人数－1”。



图 18-2 数组集合

List 接口的实现类有 ArrayList 和 LinkedList。ArrayList 是基于动态数组数据结构的实现，LinkedList 是基于链表数据结构的实现。ArrayList 访问元素速度优于 LinkedList，LinkedList 占用的内存空间比较大，但 LinkedList 在批量插入或删除数据时优于 ArrayList。

提示 List 集合关心元素是否有序，而不关心是否重复，请大家记住这个原则。例如，图 18-2 所示的班级集合中就有两个“张三”。

不同的结构对应于不同的算法，有的考虑节省占用空间，有的考虑提高运行效率，对于程序员而言，它们就像是“熊掌”和“鱼肉”，不可兼得。提高运行速度往往是以牺牲空间为代价的，而节省占用空间往往是以牺牲运行速度为代价的。

18.2.1 常用方法

List 接口继承自 Collection 接口，List 接口中的很多方法都是继承自 Collection 接口的。List 接口中常用方法如下。

1. 操作元素

- get(int index)：返回 List 集合中指定位置的元素。
- set(int index, Object element)：用指定元素替换 List 集合中指定位置的元素。
- add(Object element)：在 List 集合的尾部添加指定的元素。该方法是从 Collection 集合继承过来的。
- add(int index, Object element)：在 List 集合的指定位置插入指定元素。
- remove(int index)：移除 List 集合中指定位置的元素。
- remove(Object element)：如果 List 集合中存在指定元素，则从 List 集合中移除第一次出现的指定元素。该方法是从 Collection 集合继承过来的。
- clear()：从 List 集合中移除所有元素。该方法是从 Collection 集合继承过来的。

2. 判断元素

- isEmpty()：判断 List 集合中是否有元素，如果没有返回 true，有返回 false。该方法是从 Collection 集合继承过来的。
- contains(Object element)：判断 List 集合中是否包含指定元素，如果包含返回 true，不包含返回 false。该方法是从 Collection 集合继承过来的。

3. 查询元素

- indexOf(Object o)：从前往后查找 List 集合元素，返回第一次出现指定元素的索引，如果此列表不包含该元素，则返回－1。
- lastIndexOf(Object o)：从后往前查找 List 集合元素，返回第一次出现指定元素的索引，如果此列表不包含该元素，则返回－1。

4. 其他

- iterator()：返回迭代器(Iterator)对象，迭代器对象用于遍历集合。该方法是从 Collection 集合继承过来的。
- size()：返回 List 集合中的元素数，返回值是 int 类型。该方法是从 Collection 集合继承过来的。
- subList(int fromIndex, int toIndex)：返回 List 集合中指定的 fromIndex(包括)和 toIndex(不包括)之间的元素集合，返回值为 List 集合。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

public class HelloWorld {

    public static void main(String[] args) {

        List list = new ArrayList();                    ①

        String b = "B";

        //向集合中添加元素
        list.add("A");
        list.add(b);                                    ②
        list.add("C");
        list.add(b);                                    ③
        list.add("D");
        list.add("E");

        //打印集合元素个数
        System.out.println("集合 size = " + list.size());
        //打印集合
        System.out.println(list);
```

```
        //从前往后查找集合中的"B"元素
        System.out.println("indexOf(\"B\") = " + list.indexOf(b));
        //从后往前查找集合中的"B"元素
        System.out.println("lastIndexOf(\"B\") = " + list.lastIndexOf(b));

        //删除集合中第 1 个"B"元素
        list.remove(b);
        System.out.println("remove(3)前: " + list);
        //判断集合中是否包含"B"元素
        System.out.println("是否包含\"B\":" + list.contains(b));

        //删除集合第 4 个元素
        list.remove(3);
        System.out.println("remove(3)后: " + list);
        //判断集合是否为空
        System.out.println("list 集合是空的:" + list.isEmpty());

        System.out.println("替换前:" + list);
        //替换集合第 2 个元素
        list.set(1, "F");
        System.out.println("替换后:" + list);

        //清空集合
        list.clear();                                              ④
        System.out.println(list);


        //重新添加元素
        list.add(1);                            //发生自动装箱     ⑤
        list.add(3);

        int item = (Integer)list.get(0);        //发生自动拆箱     ⑥
    }
}
```

运行结果如下：

```
集合 size = 6
[A, B, C, B, D, E]
indexOf("B") = 1
lastIndexOf("B") = 3
remove(3)前: [A, C, B, D, E]
是否包含"B":true
remove(3)后: [A, C, B, E]
list 集合是空的:false
替换前:[A, C, B, E]
替换后:[A, F, B, E]
[]
```

代码第①行声明 List 类型集合变量 list，使用 ArrayList 类实例化 list，List 是接口，不能实例化。添加集合元素过程中可以添加重复的元素，见代码第②行和第③行。代码第④

行的 list.clear()是清空集合,但需要注意的是变量 list 所引用的对象还是存在的,不是 null,只是集合中没有了元素。

提示 在 Java 中任何集合中存放的都是对象,即引用数据类型,基本数据类型不能放到集合中。但上述代码第⑤行却将整数 1 放到集合中,这是因为这个过程中发生了自动装箱,整数 1 被封装成 Integer 对象 1,然后再放入到集合中。相反从集合中取出的也是对象,代码第⑥行从集合中取出的是 Integer 对象,之所以能够赋值给 int 类型,是因为这个过程发生了自动拆箱。

18.2.2 遍历集合

集合最常用的操作之一是遍历,遍历就是将集合中的每一个元素取出来,进行操作或计算。List 集合遍历有以下 3 种方法。

(1) 使用 for 循环遍历。List 集合可以使用 for 循环进行遍历,for 循环中有循环变量,通过循环变量可以访问 List 集合中的元素。

(2) 使用增强 for 循环遍历。增强 for 循环是针对遍历各种类型集合而推出的,推荐使用这种遍历方法。

(3) 使用迭代器遍历。Java 提供了多种迭代器,List 集合可以使用 Iterator 和 ListIterator 迭代器。

示例代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class HelloWorld {

    public static void main(String[] args) {

        List list = new ArrayList();

        String b = "B";
        //向集合中添加元素
        list.add("A");
        list.add(b);
        list.add("C");
        list.add(b);
        list.add("D");
        list.add("E");

        //1.使用 for 循环遍历
        System.out.println("--1. 使用 for 循环遍历--");
        for (int i = 0; i < list.size(); i++) {
            System.out.printf("读取集合元素(%d): %s \n", i, list.get(i));     ①
```

```
        }

        //2.使用增强 for 循环遍历
        System.out.println("--2. 使用增强 for 循环遍历--");
        for (Object item : list) {                                    ②
            String s = (String) item;                                 ③
            System.out.println("读取集合元素: " + s);
        }

        //3.使用迭代器遍历
        System.out.println("--3. 使用迭代器遍历--");
        Iterator it = list.iterator();                                ④
        while (it.hasNext()) {                                        ⑤
            Object item = it.next();                                  ⑥
            String s = (String) item;                                 ⑦
            System.out.println("读取集合元素: " + s);
        }
    }
}
```

上述代码采用3种方法遍历List集合，采用for循环遍历需要通过List集合的get方法获得元素，如代码第①行的list.get(i)。代码第②行采用增强for循环遍历list集合，从集合中取出的元素都是Object类型，代码第③行是强制转换为String类型。使用迭代器遍历，首先需要获得迭代器对象，代码第④行的list.iterator()方法可以返回迭代器对象。代码第⑤行调用迭代器hasNext()方法可以判断集合中是否还有元素可以迭代，如果有返回true，没有返回false。代码第⑥行调用迭代器的next()返回迭代的下一个元素，该方法返回的Object类型需要强制转换为String类型，见代码第⑦行。

18.3　Set 集合

Set集合是由一串无序的、不能重复的相同类型元素构成的集合。图18-3所示是一个班级的Set集合。这个Set集合中有一些学生，这些学生是无序的，不能通过类似于List集合的序号访问，而且不能有重复的学生。

提示　List集合中的元素是有序的、可重复的，而Set集合中的元素是无序的、不能重复的。List集合强调的是有序，Set集合强调的是不重复。当不考虑顺序，且没有重复元素时，Set集合和List集合是可以互相替换的。

Set接口直接实现类主要是HashSet，HashSet是基于散列表数据结构的实现。



图18-3　Set集合

18.3.1　常用方法

Set接口也继承自Collection接口，Set接口中大部分都是继承自Collection接口，这些方法如下。

1. 操作元素

- add(Object element)：在 Set 集合的尾部添加指定的元素。该方法是从 Collection 集合继承过来的。
- remove(Object element)：如果 Set 集合中存在指定元素，则从 Set 集合中移除该元素。该方法是从 Collection 集合继承过来的。
- clear()：从 Set 集合中移除所有元素。该方法是从 Collection 集合继承过来的。

2. 判断元素

- isEmpty()：判断 Set 集合中是否有元素，如果没有返回 true，否则返回 false。该方法是从 Collection 集合继承过来的。
- contains(Object element)：判断 Set 集合中是否包含指定元素，如果包含返回 true，否则返回 false。该方法是从 Collection 集合继承过来的。

3. 其他

- iterator()：返回迭代器对象，迭代器对象用于遍历集合。该方法是从 Collection 集合继承过来的。
- size()：返回 Set 集合中的元素数，返回值是 int 类型。该方法是从 Collection 集合继承过来的。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.HashSet;
import java.util.Set;

public class HelloWorld {

    public static void main(String[ ] args) {

        Set set = new HashSet();                              ①

        String b = "B";

        //向集合中添加元素
        set.add("A");
        set.add(b);                                           ②
        set.add("C");
        set.add(b);                                           ③
        set.add("D");
        set.add("E");

        //打印集合元素个数
        System.out.println("集合 size = " + set.size());      ④
        //打印集合
```

```
        System.out.println(set);
        //删除集合中"B"元素
        set.remove(b);
        //判断集合中是否包含"B"元素
        System.out.println("是否包含\"B\":" + set.contains(b));
        //判断集合是否为空
        System.out.println("set 集合是空的:" + set.isEmpty());

        //清空集合
        set.clear();
        System.out.println(set);
    }
}
```

运行结果如下：

```
集合 size = 5
[A, B, C, D, E]
是否包含"B":false
set 集合是空的:false
[]
```

代码第①行声明 Set 类型集合变量 set，使用 HashSet 类实例化 set 对象，Set 是接口，不能实例化。添加集合元素时试图添加重复的元素，见代码第②行和第③行，但是 Set 集合不能添加重复元素，所以代码第④行打印集合元素个数是 5。

18.3.2 遍历集合

Set 集合中的元素由于没有序号，所以不能使用 for 循环进行遍历，但可以使用增强 for 循环和迭代器进行遍历。事实上，这两种遍历方法也是继承自 Collection 集合，也就是说，所有的 Collection 集合类型都有这两种遍历方式。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class HelloWorld {

    public static void main(String[] args) {

        Set set = new HashSet();

        String b = "B";
```

```
            //向集合中添加元素
            set.add("A");
            set.add(b);
            set.add("C");
            set.add(b);
            set.add("D");
            set.add("E");

            //1.使用增强 for 循环遍历
            System.out.println("--1. 使用增强 for 循环遍历--");
            for (Object item : set) {
                String s = (String) item;
                System.out.println("读取集合元素: " + s);
            }

            //2.使用迭代器遍历
            System.out.println("--2. 使用迭代器遍历--");
            Iterator it = set.iterator();
            while (it.hasNext()) {
                Object item = it.next();
                String s = (String) item;
                System.out.println("读取集合元素: " + s);
            }
        }
    }
```

上述代码采用两种方法遍历 Set 集合，具体实现与 List 集合完全一样，这里不再赘述。

18.4 Map 集合

Map(映射)集合表示一种非常复杂的集合，允许按照某个键来访问元素。Map 集合由两个集合构成：一个是键(key)集合；一个是值(value)集合。键集合是 Set 类型，因此不能有重复的元素。而值集合是 Collection 类型，可以有重复的元素。Map 集合中的键和值是成对出现的。

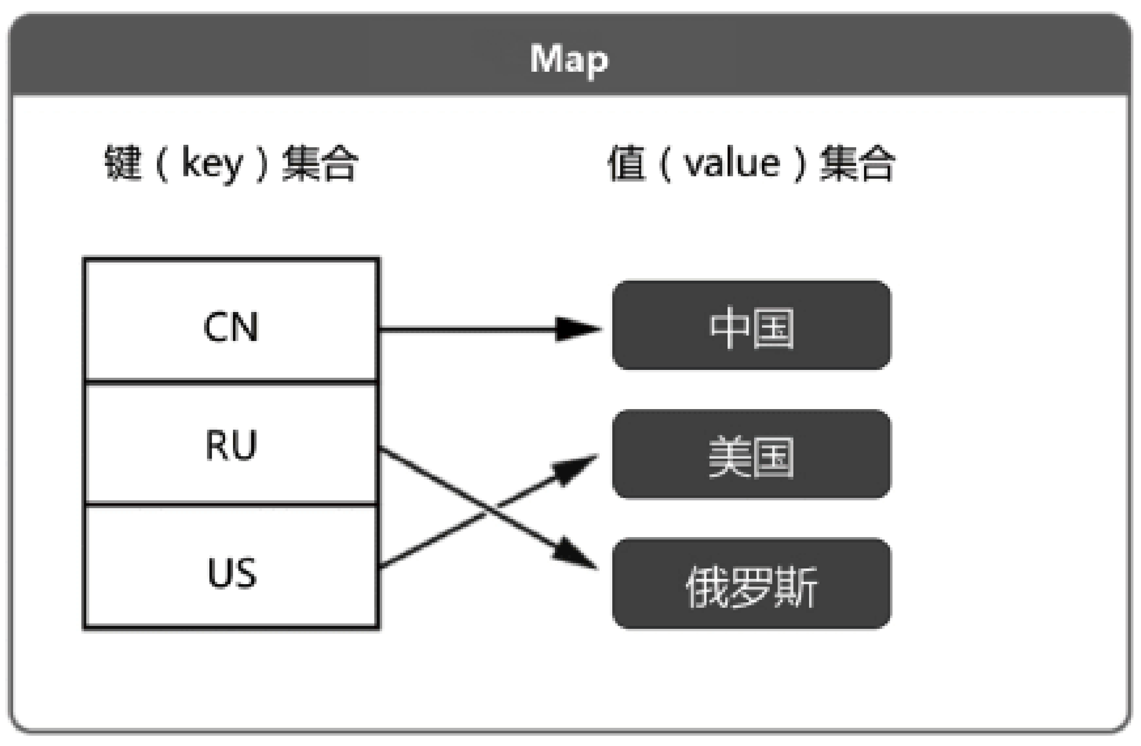


图 18-4 Map 集合

图 18-4 所示是 Map 类型的“国家代号”集合。键是国家代号集合，不能重复。值是国家集合，可以重复。

提示 Map 集合更适合通过键快速访问值，就像查英文字典一样，键就是要查的英文单词，而值是英文单词的翻译和解释等。有时，一个英文单词会对应多个翻译和解释，这是与 Map 集合特性对应的。

Map 接口直接实现类主要是 HashMap，HashMap 是基于散列表数据结构的实现。

18.4.1 常用方法

Map 集合中包含两个集合(键和值),所以操作起来比较麻烦。Map 接口提供很多方法用来管理和操作集合,主要方法如下。

1. 操作元素

- get(Object key):返回指定键所对应的值。如果 Map 集合中不包含该键值对,则返回 null。
- put(Object key, Object value):指定键值对添加到集合中。
- remove(Objectkey):移除键值对。
- clear():移除 Map 集合中所有键值对。

2. 判断元素

- isEmpty():判断 Map 集合中是否有键值对,如果没有返回 true,否则返回 false。
- containsKey(Object key):判断键集合中是否包含指定元素,如果包含返回 true,否则返回 false。
- containsValue(Object value):判断值集合中是否包含指定元素,如果包含返回 true,否则返回 false。

3. 查看集合

- keySet():返回 Map 中的所有键集合,返回值是 Set 类型。
- values():返回 Map 中的所有值集合,返回值是 Collection 类型。
- size():返回 Map 集合中键值对数。

示例代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.HashMap;
import java.util.Map;

public class HelloWorld {

    public static void main(String[] args) {

        Map map = new HashMap();                    ①

        map.put(102, "张三");
        map.put(105, "李四");                       ②
        map.put(109, "王五");
        map.put(110, "董六");
        //"李四"值重复
        map.put(111, "李四");                       ③
        //109 键已经存在,替换原来值"王五"
        map.put(109, "刘备");                       ④

        //打印集合元素个数
```

```
            System.out.println("集合 size = " + map.size());
            //打印集合
            System.out.println(map);

            //通过键取值
            System.out.println("109 - " + map.get(109));            ⑤
            System.out.println("108 - " + map.get(108));            ⑥

            //删除键值对
            map.remove(109);
            //判断键集合中是否包含 109
            System.out.println("键集合中是否包含 109:":" + map.containsKey(109));
            //判断值集合中是否包含 "李四"
            System.out.println("值集合中是否包含:" + map.containsValue("李四"));

            //判断集合是否为空
            System.out.println("集合是空的:" + map.isEmpty());

            //清空集合
            map.clear();
            System.out.println(map);
        }
    }
```

运行结果如下：

```
集合 size = 5
{102 = 张三, 105 = 李四, 109 = 刘备, 110 = 董六, 111 = 李四}
109 - 刘备
108 - null
键集合中是否包含 109:false
值集合中是否包含:true
集合是空的:false
{}
```

代码第①行声明 Map 类型集合变量 map，使用 HashMap 类实例化 map，Map 是接口，不能实例化。Map 集合添加键值对时需要注意两个问题：第一，如果键已经存在，则会替换原有值，见代码第④行是 109 键，原来对应的是"王五"，该语句会替换为"刘备"；第二，如果这个值已经存在，则不会替换，见代码第②行和第③行，添加了两个相同的值"李四"。

代码第⑤行和第⑥行是通过键取对应的值，如果不存在键值对，则返回 null，代码第⑥行的 108 键对应的值不存在，所以这里打印的是 null。

18.4.2 遍历集合

Map 集合遍历与 List 和 Set 集合不同，Map 有两个集合，因此遍历时可以只遍历值的集合，也可以只遍历键的集合，还可以同时遍历。这些遍历过程都可以使用增强 for 循环和迭代器进行遍历。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HelloWorld {

    public static void main(String[] args) {

        Map map = new HashMap();

        map.put(102, "张三");
        map.put(105, "李四");
        map.put(109, "王五");
        map.put(110, "董六");
        map.put(111, "李四");

        //1.使用增强 for 循环遍历
        System.out.println("--1. 使用增强 for 循环遍历--");
        //获得键集合
        Set keys = map.keySet();                                          ①
        for (Object key : keys) {
            int ikey = (Integer) key;                       //自动拆箱       ②
            String value = (String) map.get(ikey);        //自动装箱       ③
            System.out.printf("key=%d - value=%s \n", ikey, value);
        }

        //2.使用迭代器遍历
        System.out.println("--2. 使用迭代器遍历--");
        //获得值集合
        Collection values = map.values();                                 ④
        //遍历值集合
        Iterator it = values.iterator();
        while (it.hasNext()) {
            Object item = it.next();
            String s = (String) item;
            System.out.println("值集合元素: " + s);
        }

    }
}
```

上述代码第①行是获得键集合，返回值是 Set 类型。在遍历键时，从集合里取出的元素类型都是 Object，代码第②行是将 key 强制类型转换为 Integer，然后又赋值给 int 整数，这个过程发生了自动拆箱。代码第③行是通过键获得对应的值。代码第④行是获得值集合，

它是 Collection 类型。遍历 Collection 集合与遍历 Set 集合是一样的，这里不再赘述。

本章小结

本章介绍了 Java 中的集合，其中包括常用接口 Collection、Set、List 和 Map，重点掌握 Set、List 和 Map 3 个接口，熟悉具体实现类，熟练几种集合的遍历操作。

18.5 同步练习

1. 判断对错。

(1) 集合类型分为 Collection 和 Map。(　　)

(2) Set 里的元素是不能重复的。(　　)

(3) List 里的元素是可以重复的，可以通过下标索引。(　　)

(4) Map 集合由两个集合构成：一个是键(key)集合；一个是值(value)集合。(　　)

(5) List、Set 和 Map 接口都是继承自 Collection 接口。(　　)

2. 如果想创建 ArrayList 类的一个实例，下列哪个语句是正确的？(　　)

A. ArrayList myList=new Object()；

B. List myList=new ArrayList()；

C. ArrayList myList=new List()；

D. List myList=new List()；

第 19 章 泛　型

CHAPTER 19

Java 5 之后提供泛型(generics)支持,使用泛型可以最大限度地重用代码、保护类型的安全以及提高性能。泛型特性对 Java 影响最大的是集合框架的使用。本章详细介绍如何使用泛型。

19.1 一个问题的思考

为了理解什么是泛型,先看一个使用集合的示例,代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

public class HelloWorld {

    public static void main(String[] args) {

        List list = new ArrayList();

        //向集合中添加元素
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");

        //遍历集合
        for (Object item : list) {              ①
            Integer element = (Integer) item;   ②
            System.out.println("读取集合元素: " + element);
        }
    }
}
```

上述代码实现的功能很简单，就是将一些数据保存到集合中，然后再取出。但对于Java 5之前程序员而言，使用集合经常会面临一个很尴尬的问题：放入一种特定类型，但是取出时全部是Object类型，于是在具体使用时需要将元素转换为特定类型。上述代码第①行取出的元素是Object类型，在代码第②行需要强制类型转换。强制类型转换是有风险的，如果不进行判断就臆断进行类型转换，则会发生ClassCastException异常。本例代码第②行就发生了这个异常，JVM会抛出异常，打印出如下的异常堆栈跟踪信息：

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer
    at com.a51work6.HelloWorld.main(HelloWorld.java:23)
```

从异常堆栈跟踪信息可知，在源代码第23行试图将java.lang.String对象转换为java.lang.Integer对象。

在Java 5之前没有好的解决办法，在类型转换之前要通过instanceof运算符判断该对象是否是目标类型。而泛型的引入可以将这些运行时异常提前到编译期暴露出来，这增强了类型安全检查。

修改程序代码如下：

```
//HelloWorldGen.java 文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

public class HelloWorldGen {

    public static void main(String[] args) {

        List<String> list = new ArrayList<String>();              ①

        //向集合中添加元素
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");
        //list.add(new Date);        //发生编译错误             ②

        //遍历集合
        //使用增强 for 循环遍历
        for (String item : list) {                               ③
            //Integer element = (Integer) item;                  ④
            System.out.println("读取集合元素: " + item);
        }

    }
}
```

上述代码第①行在 List 和 ArrayList 后面添加了< String >，这就是 List 和 ArrayList 的泛型表示方式，尖括号中可以是任何的引用类型，它限定了集合中是否能存放该种类型的对象，所以代码第②行试图添加非 String 类型元素时会发生编译错误。

代码第③行从集合取出的元素就是 String 类型，所以如果在代码第④行试图转换为 Integer 则会发生编译错误。可见原本在运行时发生的异常提早暴露到编译期，使程序员提前发现问题，避免程序发布上线之后发生系统崩溃的情况。

提示 在集合中如果没有使用泛型，Eclipse 等 IDE 工具都会警告，如图 19-1 所示。可以单击这些警告，根据 Eclipse 的修订功能来修订这些警告，如图 19-2 所示，在弹出的对话框中选择“推断通用类型参数”就可以添加泛型。

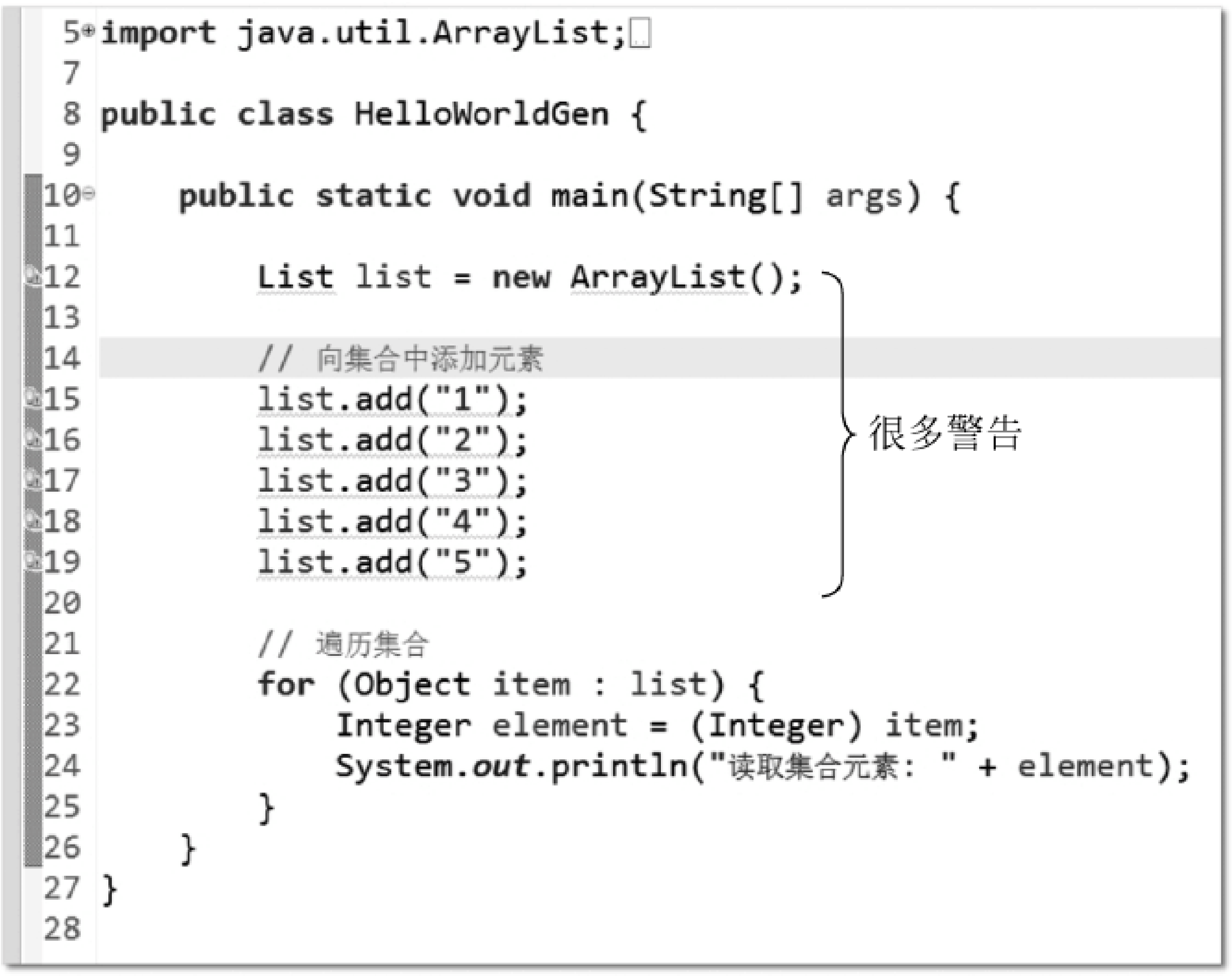


图 19-1 Eclipse 中的警告

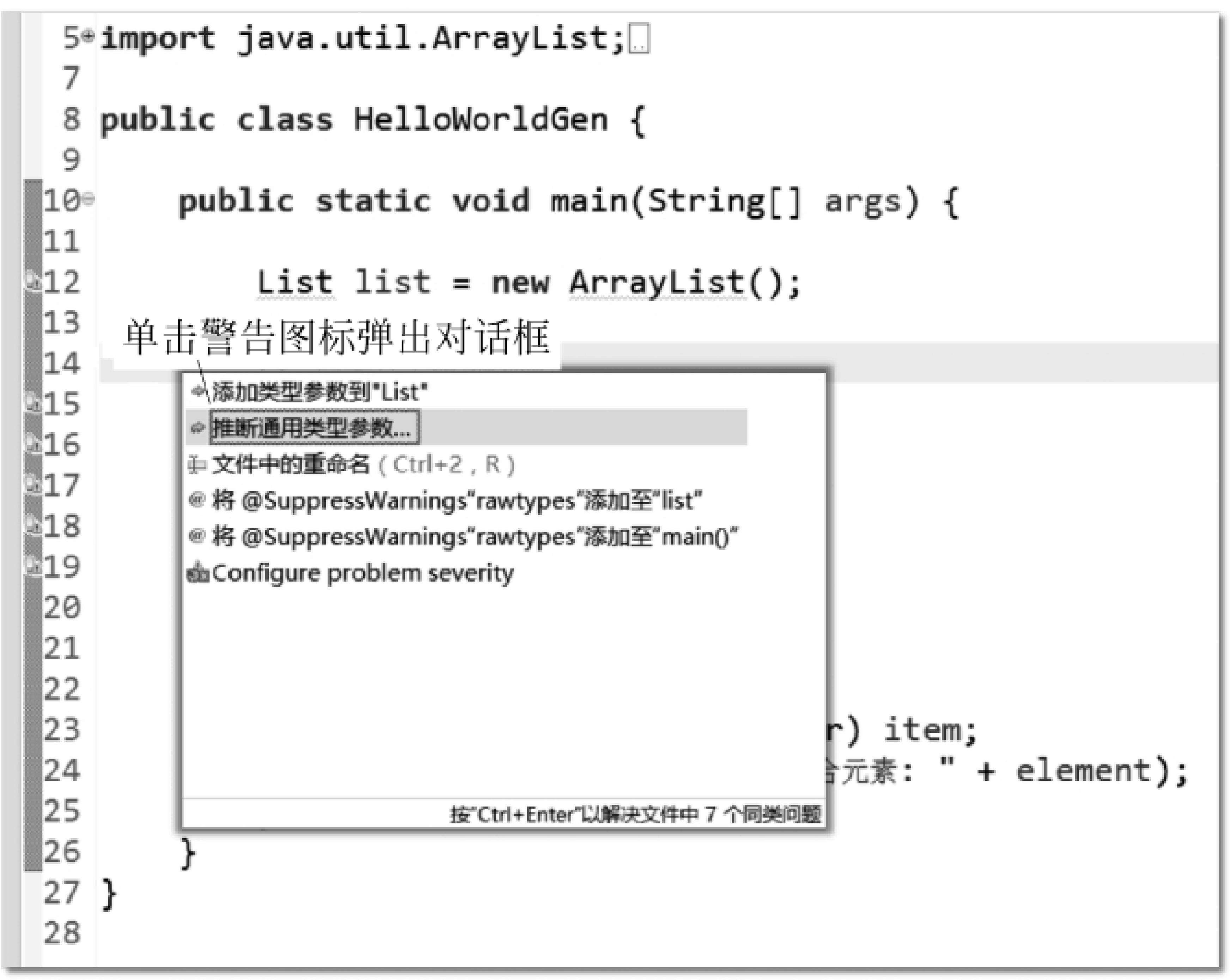


图 19-2 使用 Eclipse 中的修订功能

19.2 使用泛型

泛型对于 Java 影响最大的就是集合，Java 5 之后所有的集合类型都可以有泛型类型，可以限定存放到集合中的类型。打开如图 20-1 所示类图或打开 API 文档，会发现集合类型后面都会有< E >，如 Collection < E >、List < E >、ArrayList < E >、Set < E >和 Map < K，V >，这说明这些类型是支持泛型的。尖括号中的 E、K 和 V 等是类型参数名称，它们是实际类型的占位符。

事实上，第 18 章中所有集合示例都可以添加泛型支持。下面修改几个示例体会一下泛型的好处。先看一个 Set 泛型集合示例：

```
//HelloWorldGen.java 文件
…
//测试 Set 泛型集合方法
private static void testSet() {

    Set < String > set = new HashSet < String >();              ①
    //向集合中添加元素
    set.add("A");
    set.add("D");
    set.add("E");

    //1.使用增强 for 循环遍历
    System.out.println("--1. 使用增强 for 循环遍历--");
    for (String item : set) {                                   ②
        System.out.println("读取集合元素: " + item);
    }

    //2.使用迭代器遍历
    System.out.println("--2. 使用迭代器遍历--");
    Iterator < String > it = set.iterator();                    ③
    while (it.hasNext()) {
        String item = it.next();                                ④
        System.out.println("读取集合元素: " + item);
    }
}
```

上述代码第①行的 Set 和 HashSet 类型后面都指定了泛型，< String >说明实际类型是 String。因为有了泛型可以保证从集合中取出的元素一定是 String 类型，所以代码第②行声明元素的类型是 String。

在采用 Iterator 迭代器遍历集合时，也需要为迭代器指定泛型，限定它的实际类型是 String，见代码第③行。指定泛型的迭代器，在取出元素时不需要强制类型转换，见代

码第④行。

再看一个 Map 泛型集合示例：

```
//HelloWorldGen.java 文件
…
//测试 Map 泛型集合方法
private static void testMap() {

    Map<Integer, String> map = new HashMap<Integer, String>();      ①

    map.put(102, "张三");
    map.put(105, "李四");
    map.put(109, "王五");
    map.put(110, "董六");

    //1.使用增强 for 循环遍历
    System.out.println("--1.使用增强 for 循环遍历--");
    //获得键集合
    Set<Integer> keys = map.keySet();                               ②
    for (Integer key : keys) {                                      ③
        String value = map.get(key);                                ④
        System.out.printf("key = %d - value = %s \n", key, value);
    }

    //2.使用迭代器遍历
    System.out.println("--2. 使用迭代器遍历--");
    //获得值集合
    Collection<String> values = map.values();                       ⑤
    //遍历值集合
    Iterator<String> it = values.iterator();
    while (it.hasNext()) {
        String item = it.next();
        System.out.println("值集合元素: " + item);
    }
}
```

上述代码第①行中 Map<Integer, String>是指定 Map 泛型集合类型，其中键集合限定 Integer 类型，值集合限定 String 类型，HashMap<Integer, String>也需要同样的泛型。代码第②行是取出 Map 中键集合，需要指定它的类型是 Set<Integer>。代码第③行遍历键集合，其中取出的元素是 Integer 类型。代码第④行是从 Map 集合中取出值，它是 String 类型。这里都不需要强制类型转换，使用起来非常方便。代码第⑤行是取出 Map 中的值集合，它是 Collection<String>类型。迭代器的遍历过程与 Set 泛型集合示例类似，这里不再赘述。

19.3 自定义泛型类

根据自己的需要也可以自定义泛型类、泛型接口和带有泛型参数的方法。下面通过示例介绍泛型类。数据结构中有一种队列(queue)数据结构,如图19-3所示。它的特点是遵守先入先出(First Input First Output,FIFO)规则。



图 19-3 队列数据结构

虽然 Java SE 已经提供了支持泛型的队列 java.util.Queue<E>类型,但是为了学习泛型,本节还是要介绍一个自己实现的支持泛型的队列集合。

具体实现代码如下:

```
//Queue.java 文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

/**
 * 自定义的泛型队列集合
 */
public class Queue<T> {                                  ①

    //声明保存队列元素集合 items
    private List<T> items;                               ②

    //构造方法初始化是集合 items
    public Queue() {
        this.items = new ArrayList<T>();                 ③
    }

    /**
     * 入队方法
     * @param item 参数需要入队的元素
     */
    public void queue(T item) {                          ④
        this.items.add(item);
    }

    /**
     * 出队方法
     * @return 返回出队元素
     */
    public T dequeue(){                                  ⑤
        if (items.isEmpty()) {
            return null;
        } else {
```

```
            return this.items.remove(0);                    ⑥
        }
    }

    @Override
    public String toString() {
        return items.toString();
    }

}
```

上述代码第①行定义了 Queue<T>泛型类型的队列，T 是参数类型占位符。代码第②行是声明一个 List 泛型集合成员变量 items，用来保存队列中的元素。代码第③行是构造方法，初始化 items 成员变量。

代码第④行的 queue()方法是队列入队方法，其中参数 item 是要入队的元素，参数类型使用占位符 T 表示，注意要与 Queue<T>中的占位符保持一致。

代码第⑤行的 dequeue()是出队方法，返回出队的那个元素，返回值类型用占位符 T 表示，注意要与 Queue<T>中的占位符保持一致。在 dequeue()方法中首先判断集合是否有元素，如果没有元素，则返回 null；否则通过第⑥行的 this.items.remove(0)方法删除队列的第一个元素，并把删除的元素返回，以达到出队的目的。

提示　**泛型中参数类型占位符可以是任何大写或小写的英文字母，一般情况下习惯于使用字母 T、E、K 和 U 等大写英文字母，但也可以使用其他的字母。**

调用队列示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        Queue<String> genericQueue = new Queue<String>();       ①
        genericQueue.queue("A");
        genericQueue.queue("C");
        genericQueue.queue("B");
        genericQueue.queue("D");
        //genericQueue.queue(1);        //编译错误                ②

        System.out.println(genericQueue);
        genericQueue.dequeue();                                   ③

        System.out.println(genericQueue);

    }
}
```

输出结果如下：

```
[A, C, B, D]
[C, B, D]
```

上述代码使用了刚刚自定义的支持泛型的队列 Queue 集合，使用它与使用 Java SE 提供的泛型集合没有什么区别。首先在代码第①行实例化 Queue 对象，通过尖括号指定限定的类型是 String，这个队列中只能存放 String 类型数据。代码第②行试图向队列中添加 1，即整数数据，则会发生编译错误。代码第③行出队后操作，通过运行的结果可见，出队后第一个元素"A"会从队列中删除。

自定义泛型类时可能会用到多个类型参数，可以使用多个不同的字母作为占位符，类似于 Map<K,V>。需要注意程序代码中哪些地方是用 K 表示，哪些地方是用 V 表示。

19.4 自定义泛型接口

自定义泛型接口与自定义泛型类类似，定义的方式完全一样。下面将 19.3 节的示例修改成为队列接口，代码如下：

```
//IQueue.java 文件
package com.a51work6;

/**
 * 自定义的泛型队列集合
 */
public interface IQueue<T> {                ①

    /**
     * 入队方法
     * @param item 参数需要入队的元素
     */
    public void queue(T item);              ②

    /**
     * 出队方法
     * @return 返回出队元素
     */
    public T dequeue();                     ③

}
```

上述代码定义了支持泛型的接口。代码第①行定义了 IQueue<T>泛型接口，T 是参数类型占位符。该接口中声明两个方法，代码第②行的 queue()方法是入队方法，参数类型使用占位符 T 表示的类型。代码第③行的 dequeue()方法是出队方法，返回值类型是占位符 T 表示的类型。

实现接口 IQueue<T>的具体方式有很多，可以是 List、Set 或 Hash 等不同方式，下面给出一个基于 List 的实现方式。代码如下：

```java
//ListQueue.java 文件
package com.a51work6;

import java.util.ArrayList;
import java.util.List;

/**
 * 自定义的泛型队列集合
 */
public class ListQueue<T> implements IQueue<T> {

    //声明保存队列元素集合 items
    private List<T> items;

    //构造方法初始化是集合 items
    public ListQueue() {
        this.items = new ArrayList<T>();
    }

    /**
     * 入队方法
     *
     * @param item
     *          参数需要入队的元素
     */
    @Override
    public void queue(T item) {
        this.items.add(item);
    }

    /**
     * 出队方法
     *
     * @return 返回出队元素
     */
    @Override
    public T dequeue() {
        if (items.isEmpty()) {
            return null;
        } else {
            return this.items.remove(0);
        }
    }

    @Override
    public String toString() {
        return items.toString();
    }

}
```

上述实现代码与上一节 Queue＜T＞类很相似，只是实现了 IQueue＜T＞接口。需要注意的是，实现泛型接口的具体类也应该支持泛型，所以 Queue＜T＞中类型参数名要与 IQueue＜T＞接口中的类型参数名一致，占位符所用字母相同。

19.5 泛型方法

在方法中也可以使用泛型，即方法的参数类型或返回值类型可以用类型参数表示。假设想编写一个能够比较对象大小的方法，实现代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println(isEquals(new Integer(1), new Integer(5)));        ①
        System.out.println(isEquals(1, 5));              //发生了自动装箱      ②
        System.out.println(isEquals(new Double(1.0), new Double(1.0)));      ③
        System.out.println(isEquals(1.0, 1.0));          //发生了自动装箱      ④
        System.out.println(isEquals("A", "A"));                              ⑤
    }

    //限定类型参数为 Number
    public static <T> boolean isEquals(T a, T b) {                           ⑥
        return a.equals(b);
    }
}
```

上述代码第⑥行定义了比较方法 isEquals()，该方法可以接收两个参数，它们是任何引用类型，返回值是＜T＞，指定占位符为 T，方法中的参数类型用 T 表示。

在 main 方法中代码第①～⑤行都能够正常执行。其中代码第②行和第④行参数都是基本数据类型，它们在调用过程中发生了自动装箱，被自动转换成为对象。

另外，泛型的类型参数也可以限定一个边界。例如，比较方法 isEquals()只想用于数值对象大小的比较，实现代码如下：

```
//HelloWorldLimit.java 文件
package com.a51work6;

public class HelloWorldLimit {

    public static void main(String[] args) {

        System.out.println(isEquals(new Integer(1), new Integer(5)));
        System.out.println(isEquals(1, 5));              //发生了自动装箱
        System.out.println(isEquals(new Double(1.0), new Double(1.0)));
        System.out.println(isEquals(1.0, 1.0));          //发生了自动装箱
```

```
            //System.out.println(isEquals("A", "A"));        //编译错误          ①

        }

        //限定类型参数为 Number
        public static <T extends Number> boolean isEquals(T a, T b) {            ②
            return a.equals(b);
        }

    }
```

上述代码第②行定义泛型使用< T extends Number >语句，该语句限定类型参数只能是 Number 类型。所以代码第①行试图传递 String 类型的参数，则会发生编译错误。

本章小结

本章介绍了 Java 中的泛型技术，包括泛型概念、在集合中使用泛型、自定义泛型类、自定义泛型接口和泛型方法等。通过本章的学习，应该使用泛型的优势，并且从本章之后使用集合时，尽量使用泛型集合。

19.6　同步练习

1. 下列语句中哪些是正确的？（　　）

   A. List < String > list = new ArrayList < String >();

   B. List list = new ArrayList < String >();

   C. List < String > list = new ArrayList();

   D. List list = new ArrayList();

2. 判断对错。

（1）泛型的引入可以将这些运行时异常提前到编译期暴露出来，这增强了类型安全检查。（　　）

（2）自定义泛型类 class Queue < T > {}中 T 是参数类型占位符，也可以使用小写字母 t 表示。

第 20 章 文件管理与 I/O 流

CHAPTER 20

程序经常需要访问文件和目录，读取文件信息或写入信息到文件。在 Java 语言中对文件的读写是通过 I/O 流技术实现的。本章先介绍文件管理，然后介绍 I/O 流。

20.1 文件管理

Java 语言使用 File 类对文件和目录进行操作，查找文件时需要实现 FilenameFilter 或 FileFilter 接口。另外，读写文件内容可以通过 FileInputStream、FileOutputStream、FileReader 和 FileWriter 类实现，它们属于 I/O 流。下一节会详细介绍 I/O 流。这些类和接口全部来源于 java.io 包。

20.1.1 File 类

File 类表示一个与平台无关的文件或目录。File 类名很有欺骗性，初学者会误认为 File 对象只是一个文件，其实它也可能是一个目录。

File 类中常用的方法如下。

1. 构造方法

- File(String path)：如果 path 是实际存在的路径，则该 File 对象表示的是目录；如果 path 是文件名，则该 File 对象表示的是文件。
- File(String path, String name)：path 是路径名，name 是文件名。
- File(File dir, String name)：dir 是路径对象，name 是文件名。

2. 获得文件名

- String getName( )：获得文件的名称，不包括路径。
- String getPath( )：获得文件的路径。
- String getAbsolutePath( )：获得文件的绝对路径。
- String getParent( )：获得文件的上一级目录名。

3. 文件属性测试

- boolean exists( )：测试当前 File 对象所表示的文件是否存在。
- boolean canWrite( )：测试当前文件是否可写。
- boolean canRead( )：测试当前文件是否可读。
- boolean isFile( )：测试当前文件是否是文件。

- boolean isDirectory( )：测试当前文件是否是目录。

4. 文件操作

- long lastModified( )：获得文件最近一次修改的时间。
- long length( )：获得文件的长度，以字节为单位。
- boolean delete( )：删除当前文件。如果成功返回 true，否则返回 false。
- boolean renameTo(File dest)：将重新命名当前 File 对象所表示的文件。如果成功返回 true，否则返回 false。

5. 目录操作

- boolean mkdir( )：创建当前 File 对象指定的目录。
- String[] list()：返回当前目录下的文件和目录，返回值是字符串数组。
- String[] list(FilenameFilter filter)：返回当前目录下满足指定过滤器的文件和目录，参数是实现 FilenameFilter 接口对象，返回值是字符串数组。
- File[] listFiles()：返回当前目录下的文件和目录，返回值是 File 数组。
- File[] listFiles(FilenameFilter filter)：返回当前目录下满足指定过滤器的文件和目录，参数是实现 FilenameFilter 接口对象，返回值是 File 数组。
- File[] listFiles(FileFilter filter)：返回当前目录下满足指定过滤器的文件和目录，参数是实现 FileFilter 接口对象，返回值是 File 数组。

注意　路径中会用到路径分隔符，路径分隔符在不同平台上是有区别的，UNIX、Linux 和 macOS 中使用正斜杠(/)，而 Windows 中使用反斜杠(\)。Java 支持两种写法，但是反斜杠属于特殊字符，前面需要加转义符。例如，C:\Users\a.java 在程序代码中应该使用 C:\\Users\\a.java 表示，或表示为 C:/Users/a.java。

对目录操作有两个过滤器接口：FilenameFilter 和 FileFilter。它们都只有一个抽象方法 accept。FilenameFilter 接口中的 accept 方法如下。

- boolean accept(File dir, String name)：测试指定 dir 目录中是否包含文件名为 name 的文件。

FileFilter 接口中的 accept 方法如下。

- boolean accept(File pathname)：测试指定路径名是否应该包含在某个路径名列表中。

20.1.2　案例：文件过滤

为熟悉文件操作，本节介绍一个案例，该案例从指定的目录中列出文件信息。代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.io.File;
import java.io.FilenameFilter;
public class HelloWorld {

    public static void main(String[ ] args) {
```

```
        //用 File 对象表示一个目录,.表示当前目录
        File dir = new File("./TestDir");                ①
        //创建 HTML 文件过滤器
        Filter filter = new Filter("html");              ②

        System.out.println("HTML 文件目录: " + dir);
        //列出目录 TestDir 下,文件后缀名为 HTML 的所有文件
        String files[] = dir.list(filter);        //dir.list();
        //遍历文件列表
        for (String fileName : files) {
            //为目录 TestDir 下的文件或目录创建 File 对象
            File f = new File(dir, fileName);
            //如果该 f 对象是文件,则打印文件名
            if (f.isFile()) {
                System.out.println("文件名: " + f.getName());
                System.out.println("文件绝对路径: " + f.getAbsolutePath());
                System.out.println("文件路径: " + f.getPath());
            } else {
                System.out.println("子目录: " + f);
            }
        }

    }
}

//自定义基于文件扩展名的文件过滤器
class Filter implements FilenameFilter {                 ③

    //文件扩展名
    String extent;

    //构造方法
    Filter(String extent) {
        this.extent = extent;
    }

    @Override
    public boolean accept(File dir, String name) {       ④
        //测试文件扩展名是否为 extent 所指定的
        return name.endsWith("." + extent);
    }
}
```

上述代码第①行创建 TestDir 目录对象,"./TestDir"表示当前目录下的 TestDir 目录,还可以表示为".\\TestDir"和"TestDir"。

提示 在编程时尽量使用相对路径,尽量不要使用绝对路径。"./TestDir"就是相对路径,相对路径中会用到点(.),在目录中一个点(.)表示当前目录,两个点表示(..)表示父目录。



图 20-1 Eclipse 中的当前目录

上述代码第②行创建针对 HTML 文件的过滤器 Filter，Filter 类要求实现 FilenameFilter 接口，见代码第③行。FilenameFilter 接口要求实现抽象方法 accept，见代码第④行。在该方法中判断文件名是否为指定的扩展名结尾，若是则返回 true，否则返回 false。

注意 在 Eclipse 工具中运行 Java 程序，当前目录在哪里呢？例如"./TestDir"表示当前目录下的 TestDir 子目录，那么应该在哪里创建 TestDir 目录呢？在 Eclipse 中当前目录就是工程的根目录，如图 20-1 所示，当前目录是 Eclipse 工程根目录，子目录 TestDir 位于工程根目录下。

20.2 I/O 流简介

Java 将数据的输入输出(I/O)操作当作"流"来处理，流是一组有序的数据序列。流分为两种形式：输入流和输出流，从数据源中读取数据是输入流，将数据写入到目的地是输出流。

提示 以 CPU 为中心，从外部设备读取数据到内存，进而再读入到 CPU，这是输入(Input，I)过程；将内存中的数据写入到外部设备，这是输出(Output，O)过程。所以输入输出简称为 I/O。

20.2.1 Java 流设计理念

如图 20-2 所示，数据输入的数据源有多种形式，如文件、网络和键盘等，键盘是默认的标准输入设备。而数据输出的目的地也有多种形式，如文件、网络和控制台，控制台是默认的标准输出设备。

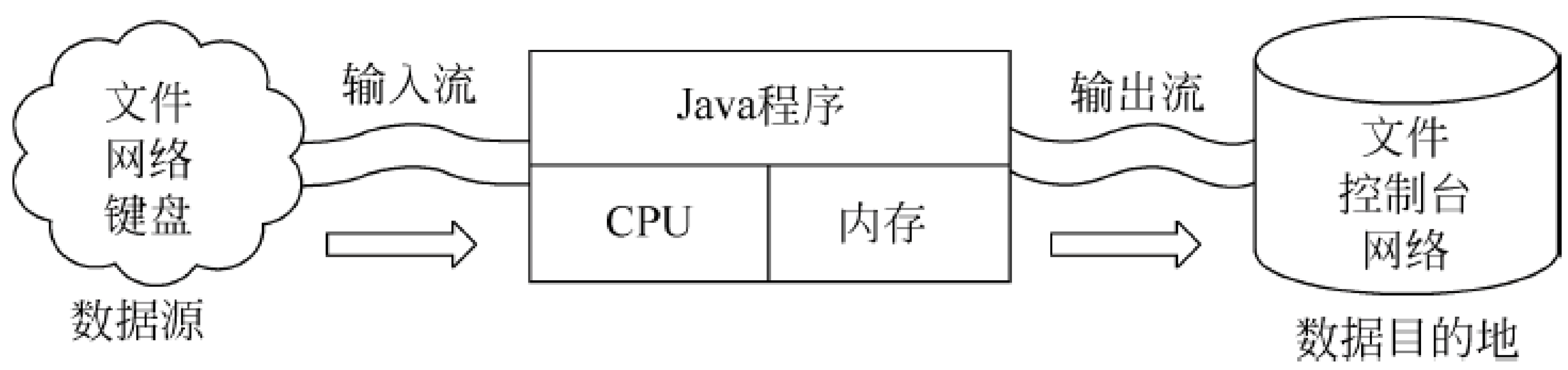


图 20-2 I/O 流

所有的输入形式都抽象为输入流，所有的输出形式都抽象为输出流，它们与设备无关。

20.2.2 流类继承层次

以字节为单位的流称为字节流，以字符为单位的流称为字符流。Java SE 提供 4 个顶级抽象类：两个字节流抽象类——InputStream 和 OutputStream；两个字符流抽象类——Reader 和 Writer。

1. 字节输入流

字节输入流根类是 InputStream，如图 20-3 所示。它有很多子类，这些类的说明如

表 20-1 所示。

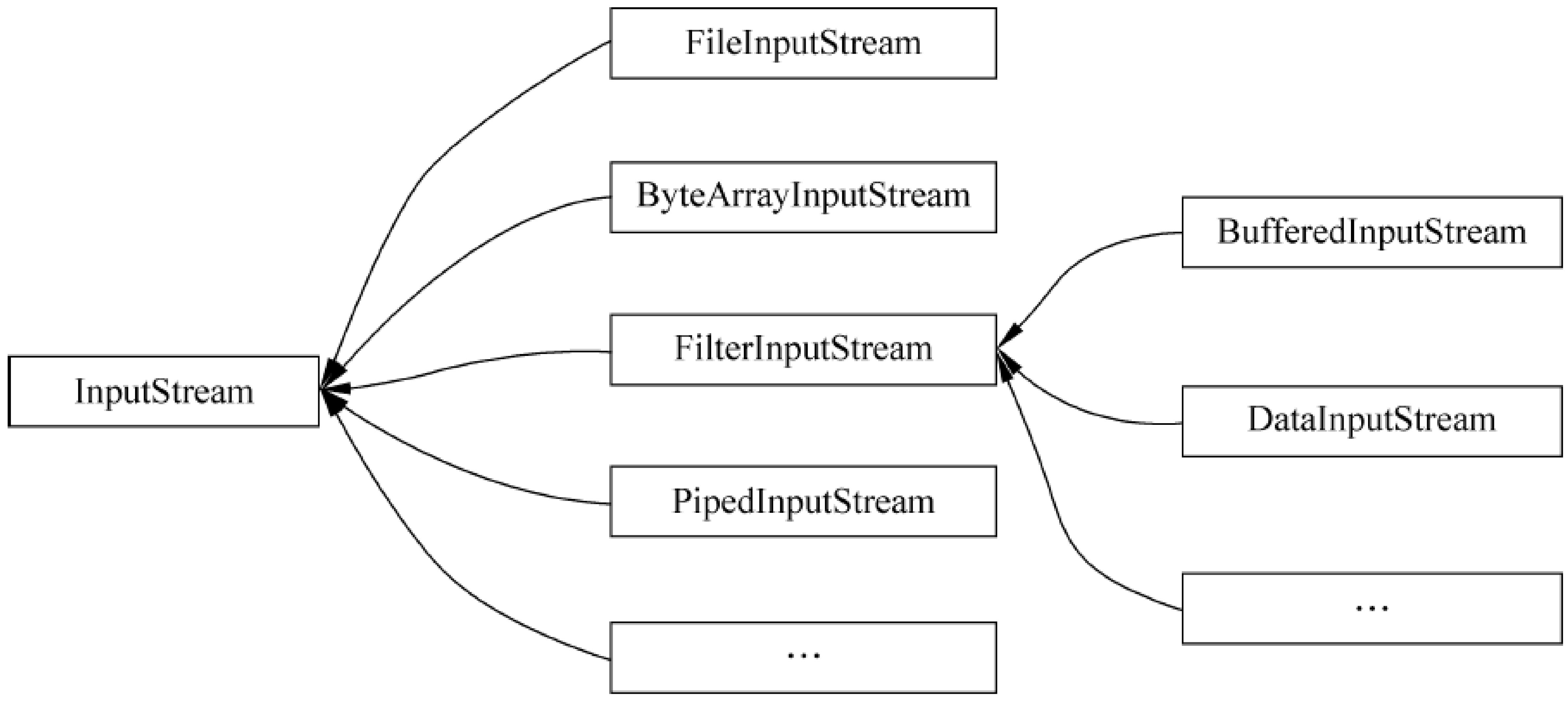


图 20-3 字节输入流类继承层次

表 20-1 主要的字节输入流

类	描 述
FileInputStream	文件输入流
ByteArrayInputStream	面向字节数组的输入流
PipedInputStream	管道输入流,用于两个线程之间的数据传递
FilterInputStream	过滤输入流,它是一个装饰器扩展其他输入流
BufferedInputStream	缓冲区输入流,它是 FilterInputStream 的子类
DataInputStream	面向基本数据类型的输入流

2. 字节输出流

字节输出流根类是 OutputStream,如图 20-4 所示。它有很多子类,这些类的说明如表 20-2 所示。

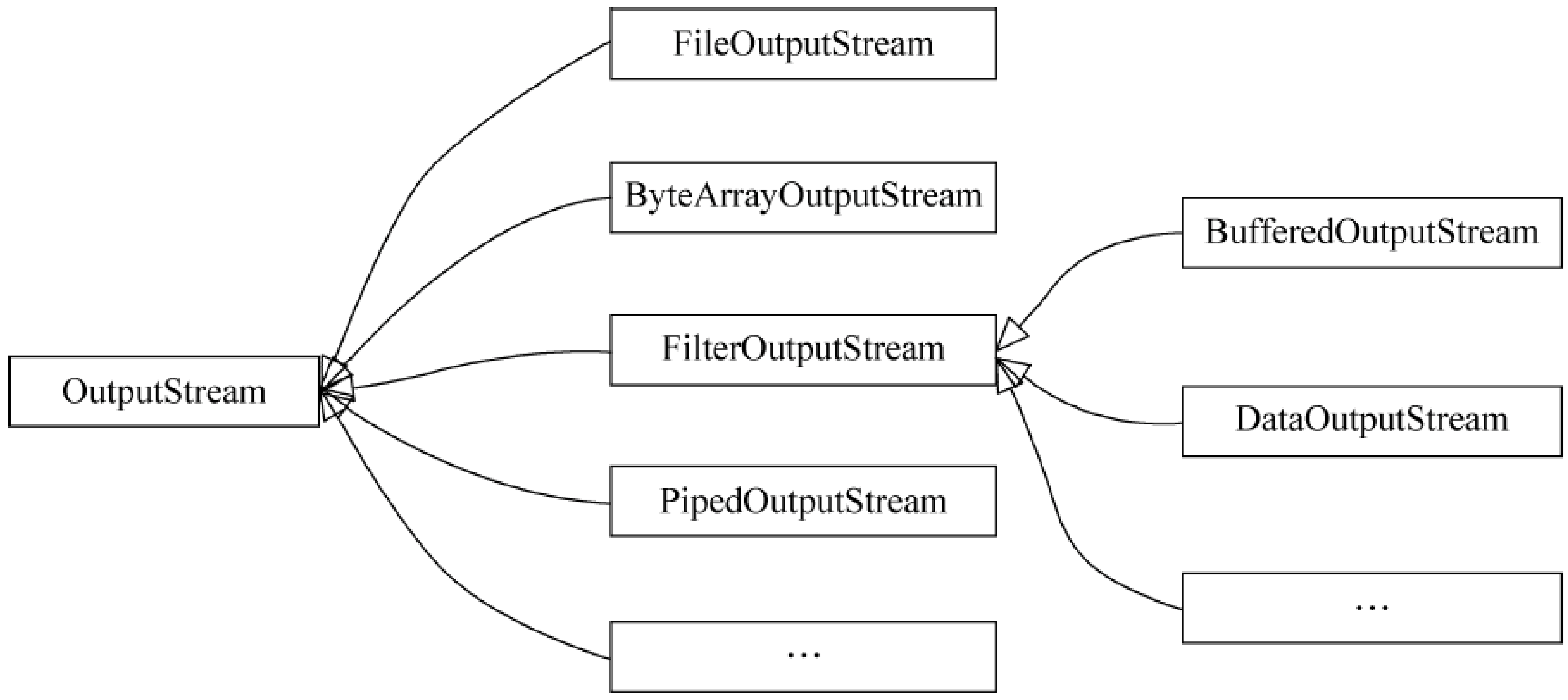


图 20-4 字节输出流类继承层次

表 20-2　主要的字节输出流

类	描　　述
FileOutputStream	文件输出流
ByteArrayOutputStream	面向字节数组的输出流
PipedOutputStream	管道输出流，用于两个线程之间的数据传递
FilterOutputStream	过滤输出流，它是一个装饰器扩展其他输出流
BufferedOutputStream	缓冲区输出流，它是 FilterOutputStream 的子类
DataOutputStream	面向基本数据类型的输出流

3. 字符输入流

字符输入流根类是 Reader，这类流以 16 位的 Unicode 编码表示的字符为基本处理单位，如图 20-5 所示。它有很多子类，这些类的说明如表 20-3 所示。

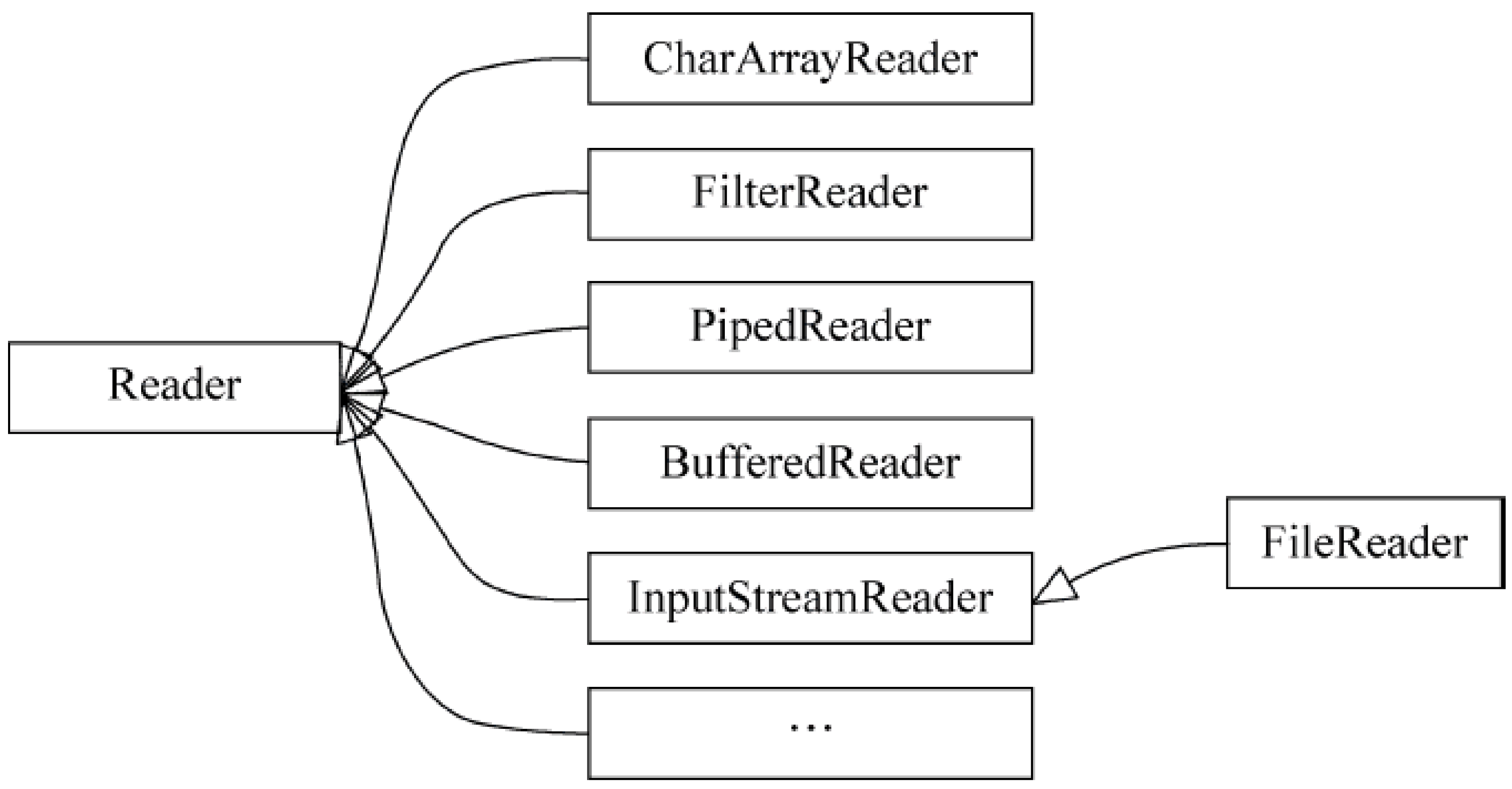


图 20-5　字符输入流类继承层次

表 20-3　主要的字符输入流

类	描　　述
FileReader	文件输入流
CharArrayReader	面向字符数组的输入流
PipedReader	管道输入流，用于两个线程之间的数据传递
FilterReader	过滤输入流，它是一个装饰器扩展其他输入流
BufferedReader	缓冲区输入流，它也是装饰器，它不是 FilterReader 的子类
InputStreamReader	把字节流转换为字符流，它也是一个装饰器，是 FileReader 的父类

4. 字符输出流

字符输出流根类是 Writer，这类流以 16 位的 Unicode 编码表示的字符为基本处理单位，如图 20-6 所示。它有很多子类，这些类的说明如表 20-4 所示。

表 20-4　主要的字符输出流

类	描　　述
FileWriter	文件输出流
CharArrayWriter	面向字符数组的输出流
PipedWriter	管道输出流，用于两个线程之间的数据传递
FilterWriter	过滤输出流，它是一个装饰器扩展其他输出流
BufferedWriter	缓冲区输出流，它也是装饰器，它不是 FilterWriter 的子类
OutputStreamWriter	把字节流转换为字符流，它也是一个装饰器，是 FileWriter 的父类

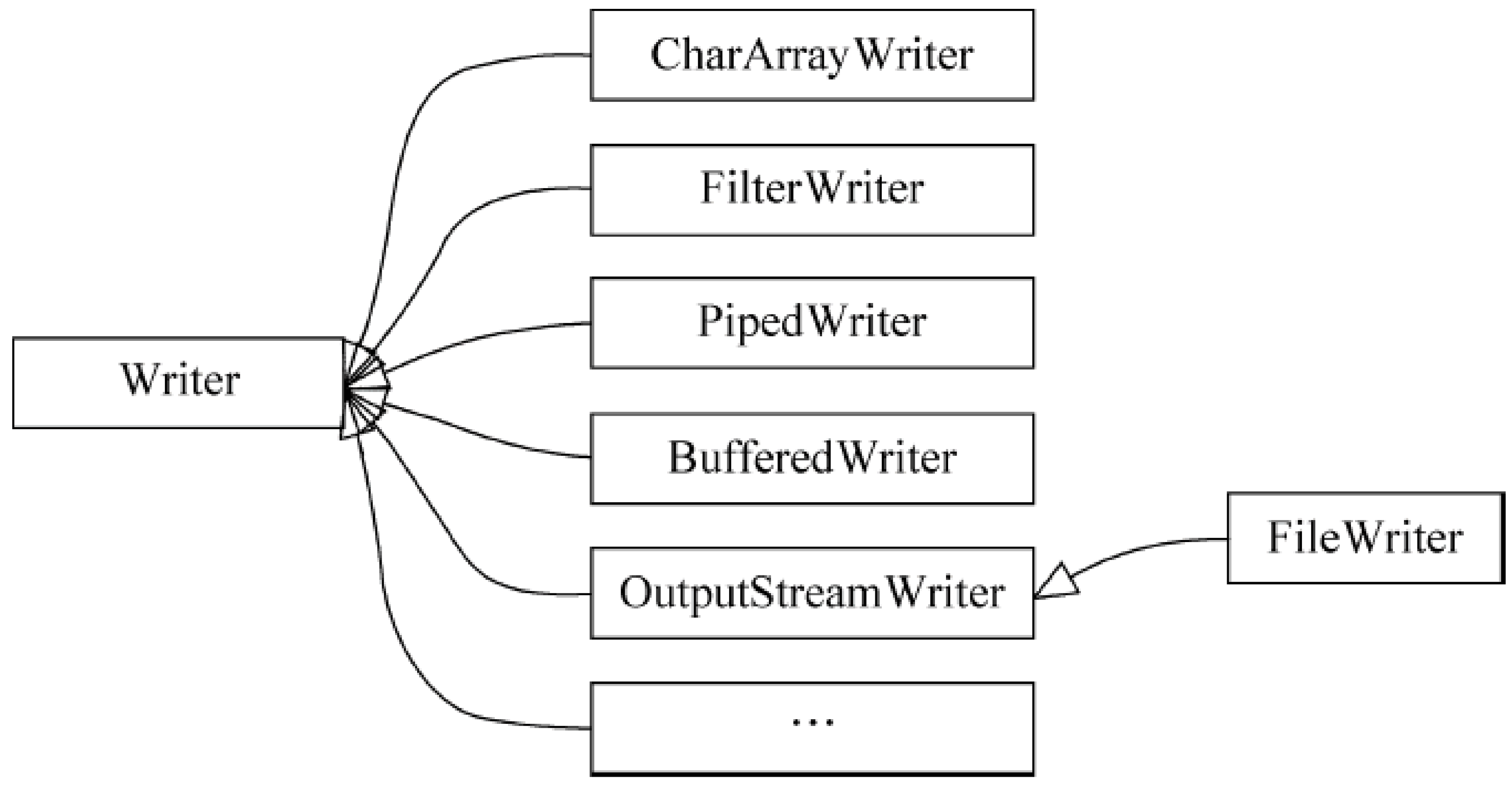


图 20-6　字符输出流类继承层次

20.3　字节流

上一节总体概述了 Java 中 I/O 流层次结构技术，本节详细介绍字节流的 API。掌握字节流的 API 先要熟悉它的两个抽象类：InputStream 和 OutputStream，了解它们有哪些主要的方法。

20.3.1　InputStream 抽象类

InputStream 是字节输入流的根类，它定义了很多方法，影响着字节输入流的行为。

InputStream 主要方法如下。

- int read()：读取 1 字节，返回 0～255 内的 int 字节值。如果已经到达流末尾，而且没有可用的字节，则返回值－1。
- int read(byte b[])：读取多字节，数据放到字节数组 b 中，返回值为实际读取的字节的数量。如果已经到达流末尾，而且没有可用的字节，则返回值－1。
- int read(byte b[ ], int off, int len)：最多读取 len 字节，数据放到以下标 off 开始的字节数组 b 中，将读取的第一个字节存储在元素 b[off]中，下一个存储在 b[off＋1]中，以此类推。返回值为实际读取的字节的数量。如果已经到达流末尾，而且没有可用的字节，则返回值－1。
- void close()：流操作完毕后必须关闭。

上述所有方法都可能会抛出 IOException，因此使用时要注意处理异常。

20.3.2　OutputStream 抽象类

OutputStream 是字节输出流的根类，它定义了很多方法，影响着字节输出流的行为。

OutputStream 主要方法如下。

- void write(int b)：将 b 写入到输出流，b 是 int 类型，占有 32 位，写入过程是写入 b 的 8个低位，b 的 24 个高位将被忽略。
- void write(byte b[ ])：将 b. length 字节从指定字节数组 b 写入到输出流。
- void write(byte b[ ], int off, int len)：把字节数组 b 中从下标 off 开始，长度为 len

的字节写入到输出流。

- void flush()：刷空输出流，并输出所有被缓存的字节。由于某些流支持缓存功能，该方法将把缓存中所有内容强制输出到流中。
- void close( )：流操作完毕后必须关闭。

上述所有方法都声明了抛出 IOException，因此使用时要注意处理异常。

注意　流（包括输入流和输出流）所占用的资源不能通过 JVM 的垃圾收集器回收，需要程序员自己释放。一种方法是可以在 finally 代码块调用 close()方法关闭流，释放流所占用的资源。另一种方法是通过自动资源管理技术管理这些流，流（包括输入流和输出流）都实现了 AutoCloseable 接口，可以使用自动资源管理技术，具体内容参考 17.4.2 节。

20.3.3　案例：文件复制

前面介绍了两种字节流常用的方法，下面通过一个案例熟悉它们的使用。该案例实现了文件复制，数据源是文件，所以会用到文件输入流 FileInputStream；数据目的地也是文件，所以会用到文件输出流 FileOutputStream。

FileInputStream 和 FileOutputStream 中主要方法都是继承自 InputStream 和 OutputStream，这在前面两节已经详细介绍，这里不再赘述。下面介绍它们的构造方法。FileInputStream 构造方法主要如下。

- FileInputStream(String name)：创建 FileInputStream 对象，name 是文件名。如果文件不存在，则抛出 FileNotFoundException 异常。
- FileInputStream(File file)：通过 File 对象创建 FileInputStream 对象。如果文件不存在，则抛出 FileNotFoundException 异常。

FileOutputStream 构造方法主要如下。

- FileOutputStream(String name)：通过指定 name 文件名创建 FileOutputStream 对象。如果 name 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。
- FileOutputStream(String name, boolean append)：通过指定 name 文件名创建 FileOutputStream 对象，append 参数如果为 true，则将字节写入文件末尾处。如果 name 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。
- FileOutputStream(File file)：通过 File 对象创建 FileOutputStream 对象。如果 file 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。
- FileOutputStream(File file, boolean append)：通过 File 对象创建 FileOutputStream 对象，append 参数如果为 true，则将字节写入文件末尾处。如果 file 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。

下面介绍如何将. /TestDir/build. txt 文件内容复制到. /TestDir/subDir/build. txt。. /TestDir/ build. txt 文件内容是 AI-162. 3764568，实现代码如下：

```
//FileCopy.java 文件
package com.a51work6;

import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopy {

    public static void main(String[] args) {

        try (FileInputStream in = new FileInputStream("./TestDir/build.txt");
                  FileOutputStream out = new FileOutputStream("./TestDir/ subDir/build.
txt")) {                                                   ①

            //准备一个缓冲区
            byte[] buffer = new byte[10];                  ②
            //首先读取一次
            int len = in.read(buffer);                     ③

            while (len != -1) {                            ④
                String copyStr = new String(buffer);       ⑤
                //打印复制的字符串
                System.out.println(copyStr);
                //开始写入数据
                out.write(buffer, 0, len);                 ⑥
                //再读取一次
                len = in.read(buffer);                     ⑦
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

控制台输出结果如下：

```
AI-162.376
456862.376
```

上述代码第①行创建 FileInputStream 和 FileOutputStream 对象，这是自动资源管理的写法，不需要自己关闭流。

第②行代码是准备一个缓冲区，它是字节数组，读取输入流的数据保存到缓冲区中，然后将缓冲区中的数据再写入到输出流中。

提示 缓冲区大小（字节数组长度）多少合适？缓冲区大小决定了一次读写操作的最多字节数，缓冲区设置的很小，会进行多次读写操作才能完成。所以如果当前计算机内存足够大，在不影响其他应用运行的情况下，缓冲区是越大越好。本例中缓冲区大小设置为10，源文件中内容是 AI-162.3764568，共有 14 个字符，由于这些字符都属于 ASCII 字符，因

此 14 个字符需要 14 字节描述，需要读写两次才能完成复制。

代码第③行是第一次从输入流中读取数据，数据保存到 buffer 中，len 是实际读取的字节数。代码第⑦行也从输入流中读取数据。由于本例中缓冲区大小设置为 10，因此两次会把数据读完，第一次读了 10 字节，第二次读了 4 字节。

代码第④行判断读取的字节数 len 是否等于－1，代码第⑦行的 len = in.read(buffer) 事实上执行了两次，第一次执行时 len 为 4，第二次执行时 len 为－1。

代码第⑤行使用字节数组构造字符串，然后通过 System.out.println(copyStr) 语句将字符串输出到控制台。从输出的结果看输出了两次，每次 10 字节，第一次输出结果 AI-162.376 容易理解，它是 AI-162.3764568 的前 10 字符；那么第二次输出的结果 456862.376 令人匪夷所思，事实上前 4 个字符(4568)是第二次读取的，后面的 6 个字符(62.376)是上一次读取的。两次读取内容如图 20-7 所示。

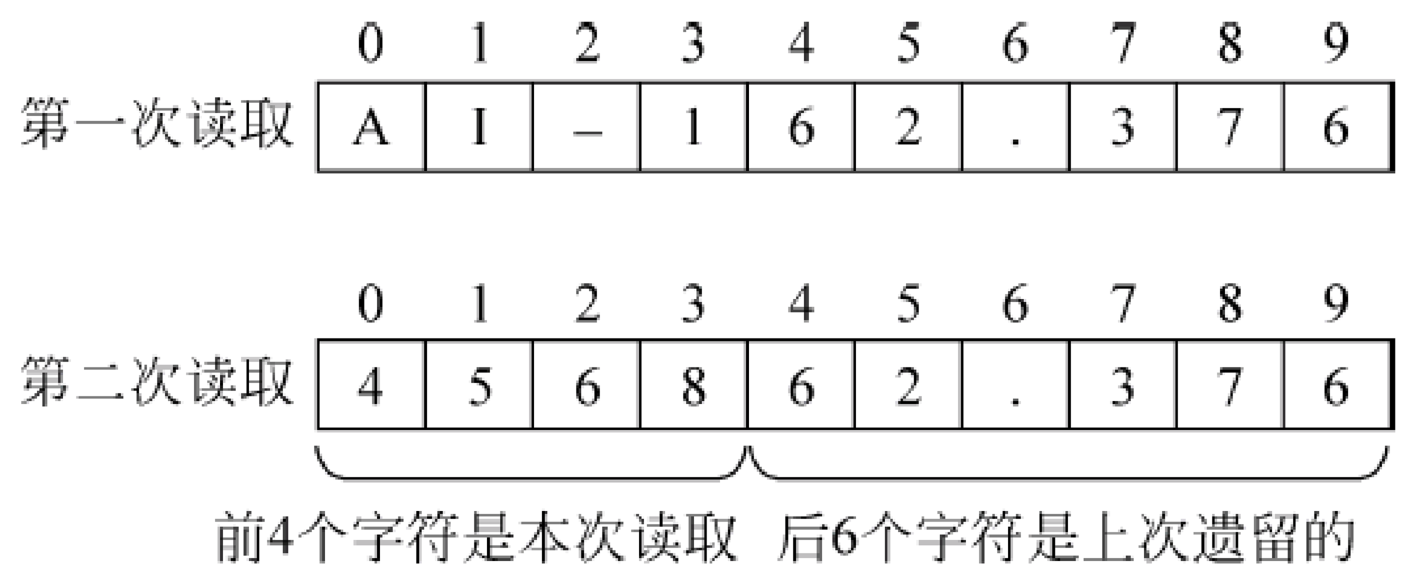


图 20-7　文件读取示意图

代码第⑥行 out.write(buffer, 0, len) 是向输出流写入数据，与读取数据对应，数据写入也调用了两次，第一次 len 为 10，将缓冲区 buffer 所有元素全部写入输出流；第二次 len 为 4，将缓冲区 buffer 所有前 4 个元素写入输出流。注意这里不要使用 void write(byte b[ ]) 方法，因为它没法控制第二次写入的字节数。

上面的案例由于使用了字节输入输出流，所以不仅可以复制文本文件，还可以复制二进制文件。

20.3.4　使用字节缓冲流

BufferedInputStream 和 BufferedOutputStream 称为字节缓冲流，使用字节缓冲流内置了一个缓冲区，第一次调用 read 方法时尽可能多地从数据源读取数据到缓冲区，后续再用 read 方法时先看看缓冲区中是否有数据，如果有则读缓冲区中的数据，否则再将数据源中的数据读入到缓冲区，这样可以减少直接读数据源的次数。通过输出流调用 write 方法写入数据时，也先将数据写入到缓冲区，缓冲区满了之后再写入数据目的地，这样可以减少直接对数据目的地写入次数。使用了缓冲字节流可以减少 I/O 操作次数，提高效率。

从图 20-3 和图 20-4 中可见，BufferedInputStream 的父类是 FilterInputStream，BufferedOutputStream 的父类是 FilterOutputStream，FilterInputStream 和 FilterOutputStream 称为过滤流。过滤流的作用是扩展其他流，增强其功能。BufferedInputStream 和 BufferedOutputStream 增强了缓冲能力。

提示　**过滤流实现了装饰器(decorator)设计模式，这种设计模式能够在运行时扩充一个类的功能。而继承在编译时扩充一个类的功能。**

BufferedInputStream 和 BufferedOutputStream 中主要方法都是继承自 InputStream 和 OutputStream，前面两节已经详细介绍了，这里不再赘述。下面介绍它们的构造方法。BufferedInputStream 构造方法主要如下。

- BufferedInputStream(InputStream in)：通过一个底层输入流 in 对象创建缓冲流对象，缓冲区大小是默认的，默认值为 8192。
- BufferedInputStream(InputStream in, int size)：通过一个底层输入流 in 对象创建缓冲流对象，size 指定缓冲区大小，缓冲区大小应该是 2 的 n 次幂，这样可提高缓冲区的利用率。

BufferedOutputStream 构造方法主要如下。

- BufferedOutputStream(OutputStream out)：通过一个底层输出流 out 对象创建缓冲流对象，缓冲区大小是默认的，默认值为 8192。
- BufferedOutputStream(OutputStream out, int size)：通过一个底层输出流 out 对象创建缓冲流对象，size 指定缓冲区大小，缓冲区大小应该是 2 的 n 次幂，这样可提高缓冲区的利用率。

下面将 20.3.3 节文件复制的案例改造成缓冲流实现。代码如下：

```
//FileCopyWithBuffer.java 文件
package com.a51work6;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopyWithBuffer {

    public static void main(String[] args) {

        try (FileInputStream fis = new FileInputStream("./TestDir/src.zip");          ①
                BufferedInputStream bis = new BufferedInputStream(fis);               ②
                FileOutputStream fos = new FileOutputStream("./TestDir/
subDir/src.zip");                                                                      ③
                BufferedOutputStream bos = new BufferedOutputStream(fos)) {           ④

            //开始时间
            long startTime = System.nanoTime();                                       ⑤
            //准备一个缓冲区
            byte[] buffer = new byte[1024];                                           ⑥
            //首先读取一次
            int len = bis.read(buffer);

            while (len != -1) {
                //开始写入数据
                bos.write(buffer, 0, len);
```

```
                //再读取一次
                len = bis.read(buffer);
            }

            //结束时间
            long elapsedTime = System.nanoTime() - startTime;        ⑦
            System.out.println("耗时: " + (elapsedTime / 1000000.0) + " 毫秒");

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行是创建文件输入流，它是一个底层流，通过它构造缓冲输入流，见代码第②行。同理，代码第④行是构造缓冲输出流。

为了记录复制过程所耗费的时间，在复制之前获取当前系统时间，见代码第⑤行，System.nanoTime()是获得当前系统时间，单位是纳秒。在复制结束之后同样获取系统时间，代码第⑦行用结束时的系统时间减去复制之前的系统时间，elapsedTime 就是耗时，但是它的单位是纳秒，需要除以 10^6 才是毫秒。

提示　在程序代码第⑥行也指定了缓冲区 buffer，这个缓冲区与缓冲流内置缓冲区不同，决定是否进行 I/O 操作次数的是缓冲流内置缓冲区，不是这个缓冲区。

为了比较，可以将 20.3.3 节案例也添加耗时输出功能。代码如下：

```
//FileCopy.java 文件
package com.a51work6;
…
public class FileCopy {

    public static void main(String[] args) {

        try (FileInputStream in = new FileInputStream("./TestDir/src.zip");
                FileOutputStream out = new FileOutputStream("./TestDir/subDir/src.zip")) {

            //开始时间，当前系统纳秒时间
            long startTime = System.nanoTime();
            //准备一个缓冲区
            byte[] buffer = new byte[1024];
            //首先读取一次
            int len = in.read(buffer);

            while (len != -1) {
                //开始写入数据
                out.write(buffer, 0, len);
```

```
                //再读取一次
                len = in.read(buffer);
            }

            //结束时间，当前系统纳秒时间
            long elapsedTime = System.nanoTime() - startTime;
            System.out.println("耗时: " + (elapsedTime / 1000000.0) + " 毫秒");

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FileCopy 与 FileCopyWithBuffer 复制相同文件 src.zip，缓冲区 buffer 都设置为 1024，那么运行的结果如下：

```
FileCopyWithBuffer 耗时: 94.927181 毫秒
FileCopy 耗时: 206.087523 毫秒
```

可能每次运行稍有不同，但是可以看出它们的差别：使用缓冲流的 FileCopyWithBuffer 明显要比不使用缓冲流的 FileCopy 速度快。

20.4 字符流

上一节介绍了字节流，本节详细介绍字符流的 API。掌握字符流的 API 先要熟悉它的两个抽象类：Reader 和 Writer，了解它们有哪些主要的方法。

20.4.1 Reader 抽象类

Reader 是字符输入流的根类，它定义了很多方法，影响着字符输入流的行为。

Reader 主要方法如下。

- int read()：读取一个字符，返回值为 0～65535(0x00～0xffff)。如果已经到达流末尾，则返回值-1。
- int read(char[] cbuf)：将字符读入到数组 cbuf 中，返回值为实际读取的字符的数量。如果已经到达流末尾，则返回值-1。
- int read(char[] cbuf, int off, int len)：最多读取 len 个字符，数据放到以下标 off 开始的字符数组 cbuf 中，将读取的第一个字符存储在元素 cbuf[off]中，下一个存储在 cbuf[off+1]中，以此类推。返回值为实际读取的字符的数量。如果已经到达流末尾，则返回值-1。
- void close()：流操作完毕后必须关闭。

上述所有方法都可能会抛出 IOException，因此使用时要注意处理异常。

20.4.2　Writer 抽象类

Writer 是字符输出流的根类，它定义了很多方法，影响着字符输出流的行为。

Writer 主要方法如下。

- void write(intc)：将整数值为 c 的字符写入到输出流，c 是 int 类型，占有 32 位，写入过程是写入 c 的 16 个低位，c 的 16 个高位将被忽略。
- void write(char[] cbuf)：将字符数组 cbuf 写入到输出流。
- void write(char[] cbuf, int off, int len)：把字符数组 cbuf 中从下标 off 开始，长度为 len 的字符写入到输出流。
- void write(String str)：将字符串 str 中的字符写入输出流。
- void write(String str,int off,int len)：将字符串 str 中从索引 off 开始处的 len 个字符写入输出流。
- void flush()：刷空输出流，并输出所有被缓存的字符。由于某些流支持缓存功能，该方法将把缓存中所有内容强制输出到流中。
- void close( )：流操作完毕后必须关闭。

上述所有方法都声明了抛出 IOException，因此使用时要注意处理异常。

注意　Reader 和 Writer 都实现了 AutoCloseable 接口，可以使用自动资源管理技术自动关闭它们。

20.4.3　案例：文件复制

前面两节介绍了字符流常用的方法，下面通过一个案例熟悉它们的使用。该案例实现了文件复制，数据源是文件，所以会用到文件输入流 FileReader；数据目的地也是文件，所以会用到文件输出流 FileWriter。

FileReader 和 FileWriter 中主要方法都是继承自 Reader 和 Writer，这在前面两节已经详细介绍，这里不再赘述。下面介绍它们的构造方法，FileReader 构造方法主要如下。

- FileReader(String fileName)：创建 FileReader 对象，fileName 是文件名。如果文件不存在，则抛出 FileNotFoundException 异常。
- FileReader(File file)：通过 File 对象创建 FileReader 对象。如果文件不存在，则抛出 FileNotFoundException 异常。

FileWriter 构造方法主要如下。

- FileWriter(String fileName)：通过指定 fileName 文件名创建 FileWriter 对象。如果 fileName 文件存在，但一个目录或文件无法打开，则抛出 FileNotFound Exception 异常。
- FileWriter(StringfileName, boolean append)：通过指定 fileName 文件名创建 FileWriter 对象，append 参数如果为 true，则将字符写入文件末尾处。如果 fileName 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。
- FileWriter(File file)：通过 File 对象创建 FileWriter 对象。如果 file 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。
- FileWriter(File file, boolean append)：通过 File 对象创建 FileWriter 对象，append

参数如果为 true，则将字符写入文件末尾处。如果 file 文件存在，但一个目录或文件无法打开，则抛出 FileNotFoundException 异常。

注意　字符文件流只能复制文本文件，不能复制二进制文件。

下面采用文件字符流重新实现 20.3.3 节文件复制案例。代码如下：

```java
//FileCopy.java 文件
package com.a51work6;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileCopy {

    public static void main(String[] args) {

        try (FileReader in = new FileReader("./TestDir/build.txt");
                FileWriter out = new FileWriter("./TestDir/subDir/build.txt")) {

            //准备一个缓冲区
            char[] buffer = new char[10];
            //首先读取一次
            int len = in.read(buffer);

            while (len != -1) {
                String copyStr = new String(buffer);
                //打印复制的字符串
                System.out.println(copyStr);
                //开始写入数据
                out.write(buffer, 0, len);
                //再读取一次
                len = in.read(buffer);
            }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

控制台输出结果如下：

```
AI-162.376
456862.376
```

上述代码与 20.3.3 节非常相似，只是将文件输入流改为 FileReader，文件输出流改为 FileWriter，缓冲区使用的是字符数组。

20.4.4 使用字符缓冲流

BufferedReader 和 BufferedWriter 称为字符缓冲流。Buffered Reader 特有方法和构造方法如下。

- String readLine()：读取一个文本行。如果已经到达流末尾，则返回值 null。
- BufferedReader(Reader in)：构造方法，通过一个底层输入流 in 对象创建缓冲流对象，缓冲区大小是默认的，默认值为 8192。
- BufferedReader(Reader in, int size)：构造方法，通过一个底层输入流 in 对象创建缓冲流对象，size 指定缓冲区大小，缓冲区大小应该是 2 的 *n* 次幂，这样可提高缓冲区的利用率。

BufferedWriter 特有方法和构造方法主要如下。

- void newLine()：写入一个换行符。
- BufferedWriter(Writer out)：构造方法，通过一个底层输出流 out 对象创建缓冲流对象，缓冲区大小是默认的，默认值为 8192。
- BufferedWriter(Writer out, int size)：构造方法，通过一个底层输出流 out 对象创建缓冲流对象，size 指定缓冲区大小，缓冲区大小应该是 2 的 *n* 次幂，这样可提高缓冲区的利用率。

下面将 20.4.3 节的文件复制的案例改造成缓冲流实现。代码如下：

```
//FileCopyWithBuffer.java 文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileCopyWithBuffer {

    public static void main(String[] args) {

        try (FileReader fis = new FileReader("./TestDir/JButton.html");
                BufferedReader bis = new BufferedReader(fis);
                FileWriter fos = new FileWriter("./TestDir/subDir/JButton
.html");
                BufferedWriter bos = new BufferedWriter(fos)) {

            //首先读取一行文本
            String line = bis.readLine();                  ①

            while (line != null) {
                //开始写入数据
                bos.write(line);                           ②
```

```
                //写一个换行符
                bos.newLine();                          ③
                //再读取一行文本
                line = bis.readLine();
            }
            System.out.println("复制完成");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行是通过字节缓冲流 readLine 方法读取一行文本，当读取文本为 null 时说明流已经读完了。代码第②行是写入文本到输出流，由于在输入流的 readLine 方法中会丢掉一个换行符或回车符，为了保持复制结果完全一样，因此需要在写完一个文本后，调用输出流的 newLine 方法写入一个换行符。

20.4.5 字节流转换字符流

有时需要将字节流转换为字符流，InputStreamReader 和 OutputStreamWriter 是为实现这种转换而设计的。

InputStreamReader 构造方法如下。

- InputStreamReader(InputStream in)：将字节流 in 转换为字符流对象，字符流使用默认字符集。
- InputStreamReader(InputStream in, StringcharsetName)：将字节流 in 转换为字符流对象，charsetName 指定字符流的字符集，字符集主要有 US-ASCII、ISO-8859-1、UTF-8 和 UTF-16。如果指定的字符集不支持，则会抛出 UnsupportedEncodingException 异常。

OutputStreamWriter 构造方法如下。

- OutputStreamWriter(OutputStream out)：将字节流 out 转换为字符流对象，字符流使用默认字符集。
- OutputStreamWriter(OutputStream out, StringcharsetName)：将字节流 out 转换为字符流对象，charsetName 指定字符流的字符集，如果指定的字符集不支持，则会抛出 UnsupportedEncodingException 异常。

下面将 20.4.3 节的文件复制的案例改造成缓冲流实现。代码如下：

```
//FileCopyWithBuffer.java 文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

public class FileCopyWithBuffer {

    public static void main(String[] args) {

        try ( //创建字节文件输入流对象
                FileInputStream fis = new FileInputStream("./TestDir/JButton.html");   ①
                //创建转换流对象
                InputStreamReader isr = new InputStreamReader(fis);
                //创建字符缓冲输入流对象
                BufferedReader bis = new BufferedReader(isr);

                //创建字节文件输出流对象
                  FileOutputStream fos = new FileOutputStream("./TestDir/subDir/JButton.
html");
                //创建转换流对象
                OutputStreamWriter osw = new OutputStreamWriter(fos);
                //创建字符缓冲输出流对象
                BufferedWriter bos = new BufferedWriter(osw)) {                       ②

            //首先读取一行文本
            String line = bis.readLine();

            while (line != null) {
                //开始写入数据
                bos.write(line);
                //写一个换行符
                bos.newLine();
                //再读取一行文本
                line = bis.readLine();
            }
            System.out.println("复制完成");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①～②行只是一条语句，将这6个流放到try(…)中，由JVM自动管理关闭。上述流从一个文件字节流构建转换流，再构建缓冲流，这个过程比较麻烦，在I/O流开发过程中经常遇到这种流的“链条”。

本章小结

本章主要介绍了 Java 文件管理和 I/O 流技术。需要熟悉 File 类使用,还需要掌握字节流两个根类 InputStream 和 OutputStream,以及字符流的两个根类 Reader 和 Writer。了解常用的装饰器流,如 InputStreamReader、OutputStreamWriter、BufferedReader、Buffered Writer、BufferedInputStream 和 BufferedOutputStream 等。

20.5 同步练习

1. 构造 BufferedInputStream 对象的合适参数是(　　)。
   A. BufferedInputStream　　B. BufferedOutputStream
   C. FileInputStream　　D. FileOuterStream
   E. File
2. 能够转换字符集的输出流的是(　　)。
   A. Java. io. InputStream　　B. Java. io. EncodedReader
   C. Java. io. InputStreamReader　　D. Java. io. InputStreamWriter
   E. Java. io. BufferedInputStream
3. 下面哪两个选项能够创建 file. txt 文件输入流?(　　)
   A. InputStream in=new FileReader("file. txt");
   B. InputStream in=new FileInputStream("file. txt");
   C. InputStream in=new InputStreamFileReader ("file. txt", "read");
   D. FileInputStream in=new FileReader(new File("file. txt"));
   E. FileInputStream in=new FileInputStream(new File("file. txt"));
4. 下面哪两个选项以追加方式创建 file. txt 文件输出流?(　　)
   A. OutputStream out=new FileOutputStream("file. txt");
   B. OutputStream out=new FileOutputStream("file. txt", "append");
   C. FileOutputStream out=new FileOutputStream("file. txt", true);
   D. FileOutputStream out=new FileOutputStream(new file("file. txt"));
   E. OutputStream out=new FileOutputStream(new File("file. txt"), true);

第21章 多线程编程

CHAPTER 21

无论PC(Personal Computer,个人计算机)还是智能手机现在都支持多任务,都能够编写并发访问程序。多线程编程可以编写并发访问程序。本章介绍多线程编程。

21.1 基础知识

那么线程究竟是什么?在Windows操作系统出现之前,PC上的操作系统都是单任务系统,只有在大型计算机上才具有多任务和分时设计。随着Windows、Linux等操作系统出现,把原本只在大型计算机中才具有的优点带到了PC系统中。

21.1.1 进程

一般可以在同一时间内执行多个程序的操作系统都有进程的概念。一个进程就是一个执行中的程序,而每一个进程都有自己独立的一块内存空间、一组系统资源。在进程的概念中,每一个进程的内部数据和状态都是完全独立的。在Windows操作系统下可以通过Ctrl+Alt+Del组合键查看进程,在UNIX和Linux操作系统下是通过ps命令查看进程的。打开Windows当前运行的进程,如图21-1所示。

在Windows操作系统中,一个进程就是一个exe或者dll程序,它们相互独立,互相也可以通信,在Android操作系统中进程间的通信应用也是很多的。

21.1.2 线程

线程与进程相似,是一段完成某个特定功能的代码,是程序中单个顺序控制的流程。但与进程不同的是,同类的多个线程共享一块内存空间和一组系统资源,所以系统在各个线程之间切换时,开销要比进程小得多,正因如此,线程被称为轻量级进程。一个进程中可以包含多个线程。

21.1.3 主线程

Java程序至少会有一个线程,这就是主线程,程序启动后由JVM创建主线程,程序结束时由JVM停止主线程。主线程负责管理子线程,即子线程的启动、挂起、停止等操作。图21-2所示是进程、主线程和子线程的关系,其中主线程负责管理子线程,即子线程的启动、挂起、停止等操作。

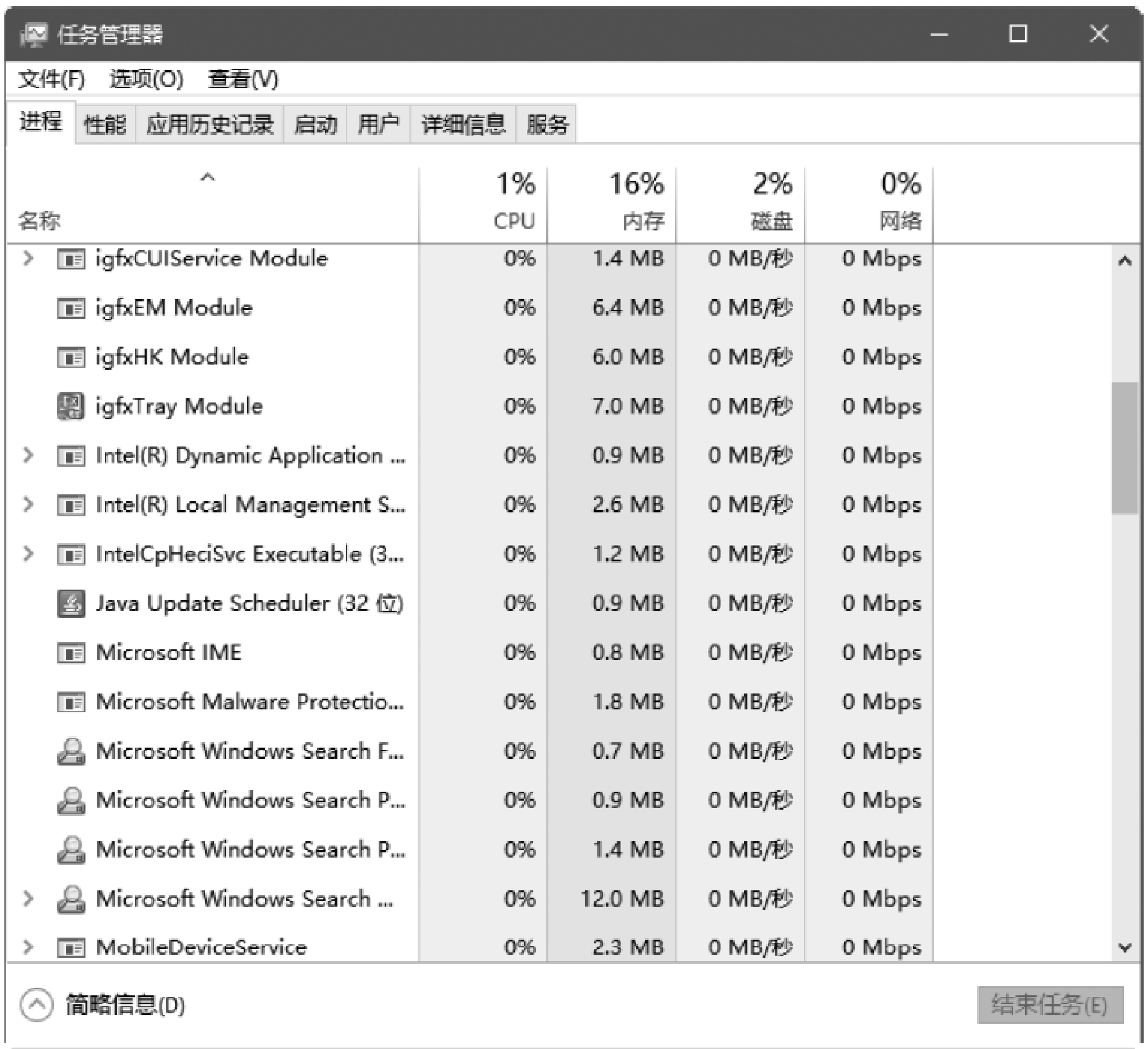


图 21-1　Windows 操作系统进程

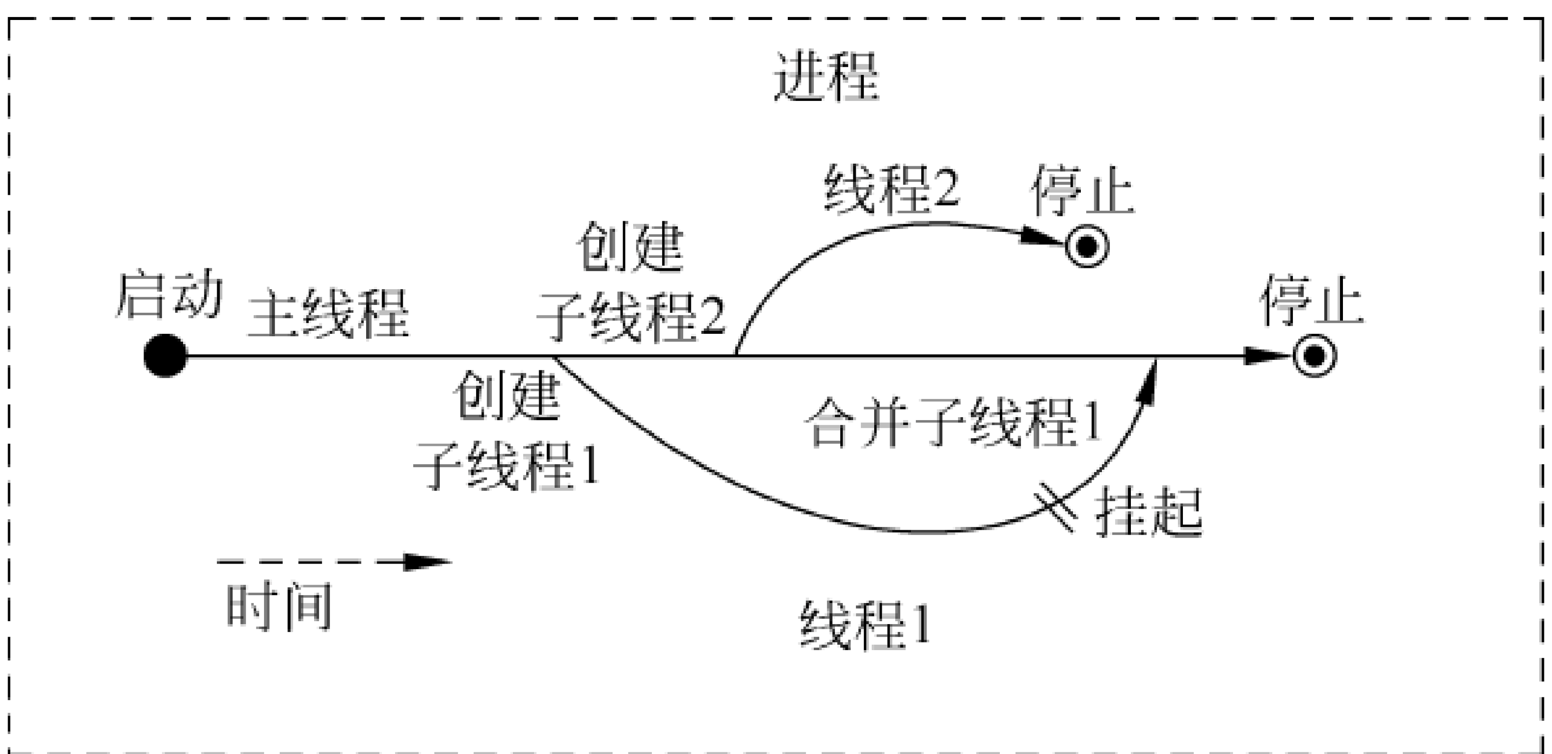


图 21-2　进程、主线程和子线程关系

获取主线程示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {
        //获取主线程
        Thread mainThread = Thread.currentThread();                    ①

        System.out.println("主线程名：" + mainThread.getName());  ②
    }
}
```

上述代码第①行的 Thread.currentThread()获得当前线程,由于在 main()方法中当前线程就是主线程,Thread 是 Java 线程类,位于 java.lang 包中。代码第②行的 getName()方法获得线程的名字,主线程名是 main,由 JVM 分配。

21.2 创建子线程

Java 中创建一个子线程涉及 java.lang.Thread 类和 java.lang.Runnable 接口。Thread 是线程类,创建一个 Thread 对象就会产生一个新的线程。而线程执行的程序代码是在实现 Runnable 接口对象的 run()方法中编写的,实现 Runnable 接口对象是线程执行对象。

线程执行对象实现 Runnable 接口的 run()方法,run()方法是线程执行的入口,该线程要执行程序代码都是在此编写的,run()方法称为线程体。

提示 **主线程中执行入口是 main(String[] args)方法,这里可以控制程序的流程,管理其他的子线程等。子线程执行入口是线程执行对象(实现 Runnable 接口对象)的 run()方法,在这个方法中可以编写子线程相关处理代码。**

21.2.1 实现 Runnable 接口

创建线程 Thread 对象时,可以将线程执行对象传递给它,这需要使用 Thread 类的两个构造方法。

- Thread(Runnable target, String name):target 是线程执行对象,实现 Runnable 接口,name 为线程名字。
- Thread(Runnable target):target 是线程执行对象,实现 Runnable 接口,线程名字是由 JVM 分配的。

下面看一个具体示例,实现 Runnable 接口的线程执行对象 Runner 代码如下:

```
//Runner.java 文件
package com.a51work6;

//线程执行对象
public class Runner implements Runnable {                          ①

    //编写执行线程代码
    @Override
    public void run() {                                            ②
        for (int i = 0; i < 10; i++) {
            //打印次数和线程的名字
            System.out.printf("第 %d 次执行 - %s\n", i,
                        Thread.currentThread().getName());         ③
            try {
                //随机生成休眠时间
                long sleepTime = (long) (1000 * Math.random());
                //线程休眠
                Thread.sleep(sleepTime);                           ④
```

```
                } catch (InterruptedException e) {
                }
            }
            //线程执行结束
            System.out.println("执行完成! " + Thread.currentThread().getName());
        }
    }
```

上述代码第①行声明实现 Runnable 接口，这要覆盖代码第②行的 run()方法，run()方法是线程体，在该方法中编写自己的线程处理代码。

本例线程体中进行了 10 次循环，每次让当前线程休眠一段时间。其中代码第③行是打印次数和线程的名字，Thread. currentThread()可以获得当前线程对象，getName()是 Thread 类的实例方法，可以获得线程的名字。代码第④行的 Thread. sleep(sleepTime)是休眠当前线程。sleep 是静态方法，它有以下两个版本。

- static void sleep(long millis)：在指定的毫秒数内让当前正在执行的线程休眠。
- static void sleep(long millis, int nanos)：在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠。

测试程序 HelloWorld 代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //创建线程 t1,参数是一个线程执行对象 Runner
        Thread t1 = new Thread(new Runner());                   ①
        //开始线程 t1
        t1.start();                                             ②

        //创建线程 t2,参数是一个线程执行对象 Runner
        Thread t2 = new Thread(new Runner(), "MyThread");       ③
        //开始线程 t2
        t2.start();                                             ④
    }
}
```

上述代码创建了两个子线程，见代码第①行和第③行，构造方法参数是线程执行对象 Runner，其中代码第①行的构造方法没有指定线程的名字，代码第③行的构造方法指定了线程名字。线程创建完成还需要调用 start()方法才能执行，见代码第②行和第④行，start()方法一旦调用，线程进入可以执行状态，可以执行状态下的线程等待 CPU 调度执行，CPU 调用后线程进入执行状态，运行 run()方法。

运行结果如下：

```
第 0 次执行 - MyThread
第 0 次执行 - Thread-0
第 1 次执行 - Thread-0
第 1 次执行 - MyThread
第 2 次执行 - MyThread
第 2 次执行 - Thread-0
第 3 次执行 - MyThread
第 3 次执行 - Thread-0
第 4 次执行 - Thread-0
第 5 次执行 - Thread-0
第 6 次执行 - Thread-0
第 4 次执行 - MyThread
第 7 次执行 - Thread-0
第 5 次执行 - MyThread
第 8 次执行 - Thread-0
第 6 次执行 - MyThread
第 9 次执行 - Thread-0
第 7 次执行 - MyThread
执行完成! Thread-0
第 8 次执行 - MyThread
第 9 次执行 - MyThread
执行完成! MyThread
```

提示　仔细分析一下运行结果，会发现两个线程是交错运行的，感觉就像是两个线程在同时运行。但是实际上一台PC通常就只有一个CPU，在某个时刻只能是一个线程在运行，而Java语言在设计时就充分考虑到线程的并发调度执行。对于程序员来说，在编程时要注意给每个线程执行的时间和机会，主要是通过让线程休眠的办法(调用sleep()方法)来让当前线程暂停执行，然后由其他线程来争夺执行的机会。如果上面的程序中没有用到sleep()方法，则就是第一个线程先执行完毕，然后第二个线程再执行完毕。所以用活sleep()方法是多线程编程的关键。

21.2.2　继承Thread线程类

事实上，Thread类也实现了Runnable接口，所以Thread类也可以作为线程执行对象，这需要继承Thread类覆盖run()方法。

采用继承Thread类重新实现21.2.1节示例。自定义线程类MyThread代码如下：

```
//MyThread.java 文件
package com.a51work6;

//线程执行对象
public class MyThread extends Thread {

    public MyThread() {                         ①
        super();                                ②
    }
```

```
        public MyThread(String name) {                  ③
            super(name);                                ④
        }

        //编写执行线程代码
        @Override
        public void run() {                             ⑤
            for (int i = 0; i < 10; i++) {
                //打印次数和线程的名字
                System.out.printf("第 %d 次执行 - %s\n", i, getName());

                try {
                    //随机生成休眠时间
                    long sleepTime = (long) (1000 * Math.random());
                    //线程休眠
                    sleep(sleepTime);
                } catch (InterruptedException e) {
                }
            }
            //线程执行结束
            System.out.println("执行完成! " + getName());
        }
    }
```

上述代码第①行和第③行定义了一个构造方法，通过 super 调用父类 Thread 构造方法。这两个 Thread 类构造方法如下。

- Thread(String name)：name 为线程指定一个名字，代码第④行调用就是此构造方法。
- Thread()：线程名字是 JVM 分配的，代码第②行调用就是此构造方法。

代码第⑤行覆盖 Thread 类的 run()方法，run()方法是线程体，需要线程执行的代码编写在这里。

测试程序 HelloWorld 代码如下：

```
    //HelloWorld.java 文件
    package com.a51work6;

    public class HelloWorld {

        public static void main(String[] args) {

            //创建线程 t1
            Thread t1 = new MyThread();                 ①
            //开始线程 t1
            t1.start();

            //创建线程 t2
            Thread t2 = new MyThread("MyThread");       ②
            //开始线程 t2
```

```
            t2.start();
        }
    }
```

代码第①行调用无参数构造方法创建线程对象t1,代码第②行调用有一个字符串参数的构造方法创建线程对象t2。

提示　由于Java只支持单重继承,继承Thread类的方式不能再继承其他父类。当开发一些图形界面的应用时,需要一个类既是一个窗口(继承JFrame)又是一个线程体,此时只能采用实现Runnable接口方式。

21.2.3　使用匿名内部类和Lambda表达式实现线程体

如果线程体使用的地方不是很多,可以不用单独定义一个类,使用匿名内部类或Lambda表达式直接实现Runnable接口。Runnable中只有一个方法是函数式接口,可以使用Lambda表达式。

重新实现21.2.1节示例。代码如下:

```
//HelloWorld.java文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

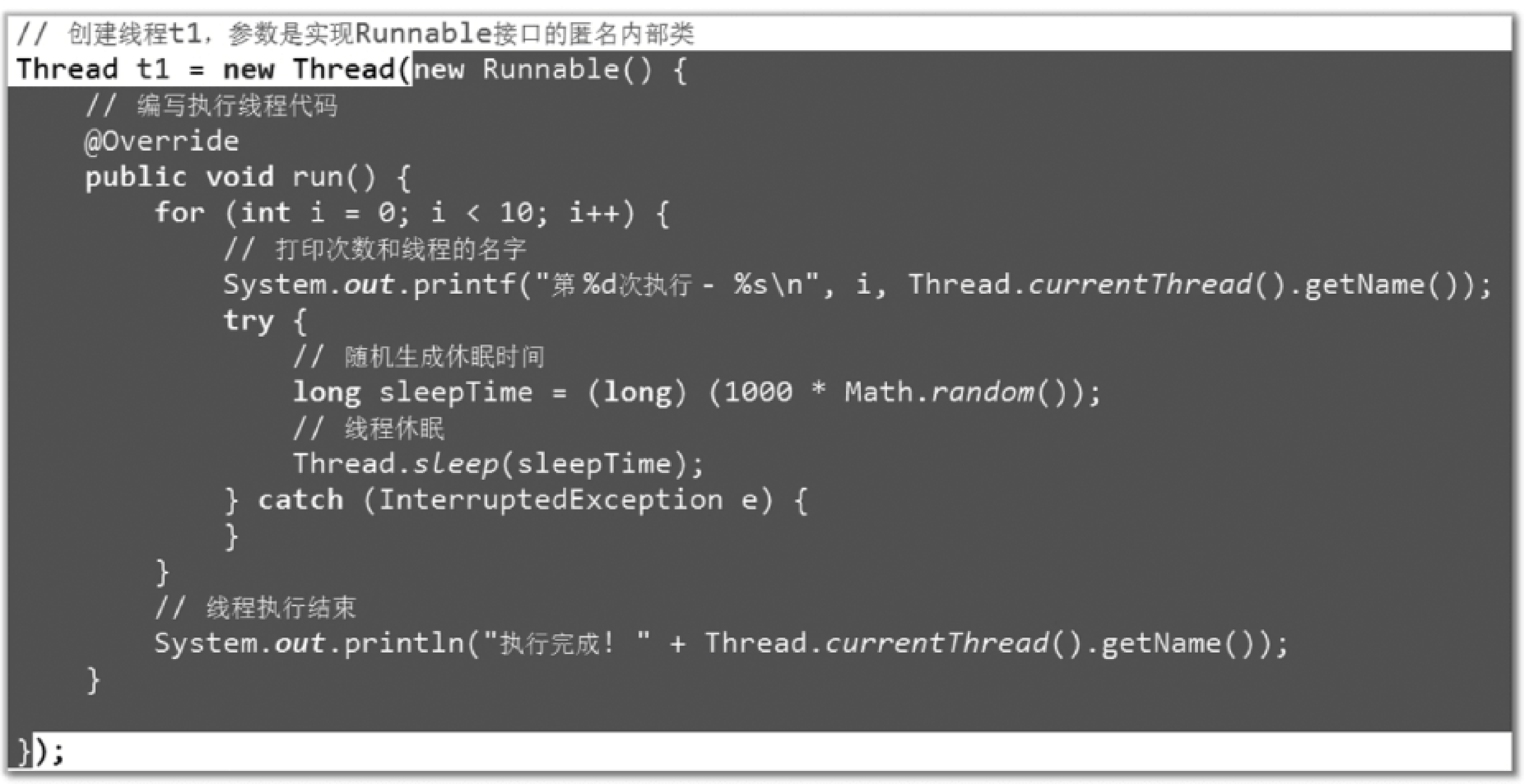
        //创建线程t1,参数是实现Runnable接口的匿名内部类
        Thread t1 = new Thread(new Runnable() {                    ①
            //编写执行线程代码
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    //打印次数和线程的名字
                    System.out.printf("第 %d次执行 - %s\n", i, Thread
.currentThread().getName());
                    try {
                        //随机生成休眠时间
                        long sleepTime = (long) (1000 * Math.random());
                        //线程休眠
                        Thread.sleep(sleepTime);
                    } catch (InterruptedException e) {
                    }
                }
                //线程执行结束
                System.out.println("执行完成! " + Thread.currentThread()
.getName());
            }

        });
```

```
        //开始线程 t1
        t1.start();

        //创建线程 t2,参数是实现 Runnable 接口的 Lambda 表达式
        Thread t2 = new Thread(() -> {                          ②
            for (int i = 0; i < 10; i++) {
                //打印次数和线程的名字
                System.out.printf("第 %d 次执行 - %s\n", i, Thread
.currentThread().getName());
                try {
                    //随机生成休眠时间
                    long sleepTime = (long) (1000 * Math.random());
                    //线程休眠
                    Thread.sleep(sleepTime);
                } catch (InterruptedException e) {
                }
            }
            //线程执行结束
            System.out.println("执行完成! " + Thread.currentThread()
.getName());
        }, "MyThread");
        //开始线程 t2
        t2.start();
    }
}
```

上述代码第①行采用匿名内部类实现 Runnable 接口,覆盖 run()方法。这里使用的是 Thread(Runnable target)构造方法。代码第②行采用 Lambda 表达式实现 Runnable 接口,覆盖 run()方法。这里使用的是 Thread(Runnable target, String name)构造方法,Lambda 表达式是它的第一个参数。匿名内部类和 Lambda 表达式代码虽然很多,但是它只是一个参数,实现了 Runnable 接口线程执行对象。如图 21-3 所示深颜色部分是匿名内部类,如图 21-4 所示深颜色部分是 Lambda 表达式。



```
// 创建线程t1,参数是实现Runnable接口的匿名内部类
Thread t1 = new Thread(new Runnable() {
    // 编写执行线程代码
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            // 打印次数和线程的名字
            System.out.printf("第 %d次执行 - %s\n", i, Thread.currentThread().getName());
            try {
                // 随机生成休眠时间
                long sleepTime = (long) (1000 * Math.random());
                // 线程休眠
                Thread.sleep(sleepTime);
            } catch (InterruptedException e) {
            }
        }
        // 线程执行结束
        System.out.println("执行完成! " + Thread.currentThread().getName());
    }

});
```

图 21-3　匿名内部类

```
// 创建线程t2，参数是实现Runnable接口的Lambda不表达式
Thread t2 = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        // 打印次数和线程的名字
        System.out.printf("第%d次执行 - %s\n", i, Thread.currentThread().getName());
        try {
            // 随机生成休眠时间
            long sleepTime = (long) (1000 * Math.random());
            // 线程休眠
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
        }
    }
    // 线程执行结束
    System.out.println("执行完成！ " + Thread.currentThread().getName());
}, "MyThread");
```

图 21-4 Lambda 表达式

提示 匿名内部类和 Lambda 表达式不需要定义一个线程类文件，使用起来很方便。特别是 Lambda 表达式使代码变得非常简洁。但是客观上匿名内部类和 Lambda 表达式会使代码可读性变差，对于初学者不容易理解。

21.3 线程状态

在线程的生命周期中，线程有 5 种状态，如图 21-5 所示。下面分别介绍。

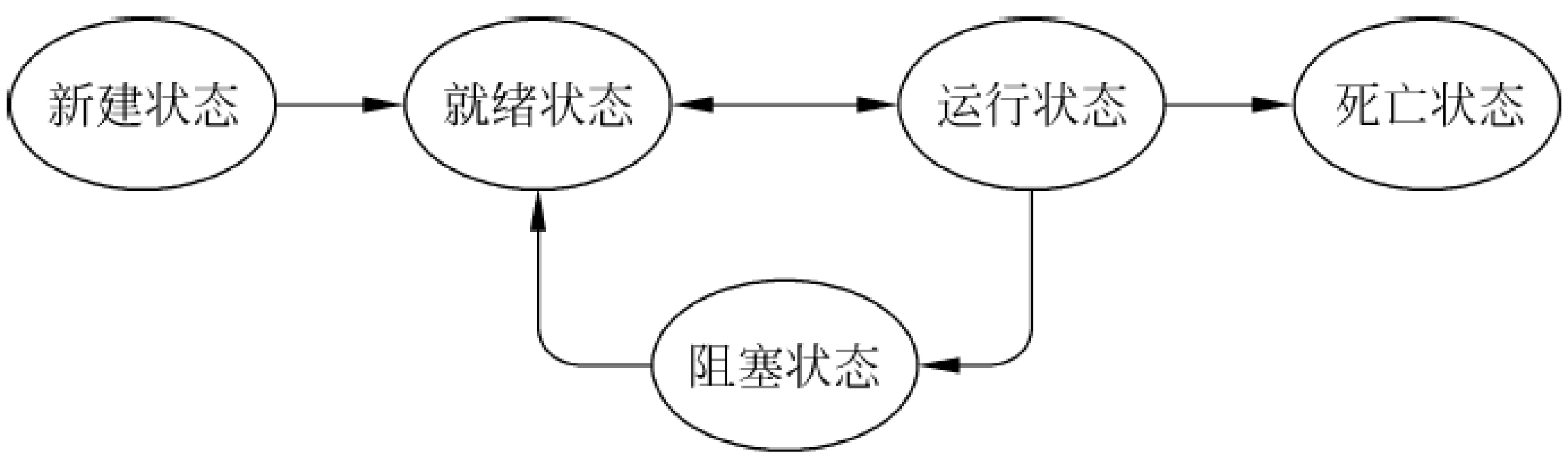


图 21-5 线程状态

1. 新建状态

新建状态(new)是通过 new 等方式创建线程对象，它仅仅是一个空的线程对象。

2. 就绪状态

当主线程调用新建线程的 start()方法后，它就进入就绪状态(runnable)。此时的线程尚未真正开始执行 run()方法，它必须等待 CPU 的调度。

3. 运行状态

CPU 调度就绪状态的线程，线程进入运行状态(running)，处于运行状态的线程独占 CPU，执行 run()方法。

4. 阻塞状态

因为某种原因运行状态的线程会进入不可运行状态，即阻塞状态(blocked)，处于阻塞状态的线程 JVM 系统不能执行，即使 CPU 空闲，也不能执行该线程。如下几个原因会导致线程进入阻塞状态。

- 当前线程调用 sleep()方法，进入休眠状态。

- 被其他线程调用了 join()方法，等待其他线程结束。
- 发出 I/O 请求，等待 I/O 操作完成期间。
- 当前线程调用 wait()方法。

处于阻塞状态可以重新回到就绪状态，如休眠结束、其他线程加入、I/O 操作完成、调用 notify 或 notifyAll 唤醒 wait 线程。

5. 死亡状态

线程退出 run()方法后就会进入死亡状态(dead)。线程进入死亡状态有可能是正常执行完成 run()方法后进入的，也有可能是由于发生异常而进入的。

21.4 线程管理

线程管理是比较头痛的事情，这是学习线程的难点。

21.4.1 线程优先级

线程的调度程序根据线程决定每次线程应当何时运行，Java 提供了 10 种优先级，分别用整数 1～10 表示，最高优先级是 10，用常量 MAX_PRIORITY 表示；最低优先级是 1，用常量 MIN_PRIORITY 表示；默认优先级是 5，用常量 NORM_PRIORITY 表示。

Thread 类提供了 setPriority(int newPriority)方法用以设置线程优先级，通过 getPriority()方法可以获得线程优先级。

设置线程优先级示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String[] args) {

        //创建线程 t1,参数是一个线程执行对象 Runner
        Thread t1 = new Thread(new Runner());
        t1.setPriority(Thread.MAX_PRIORITY);                    ①
        //开始线程 t1
        t1.start();

        //创建线程 t2,参数是一个线程执行对象 Runner
        Thread t2 = new Thread(new Runner(), "MyThread");
        t2.setPriority(Thread.MIN_PRIORITY);                    ②
        //开始线程 t2
        t2.start();
    }
}
```

在代码第①行设置线程 t1 优先级最高，代码第②行设置线程 t2 优先级最低。

提示　多次运行上面的示例会发现，t1线程经常先运行，但是偶尔t2线程也会先运行。这些现象说明，影响线程获得CPU时间的因素除了线程优先级外，还与操作系统有关。

21.4.2　等待线程结束

在介绍线程状态时提到过join()方法，当前线程调用t1线程的join()方法，则阻塞当前线程，等待t1线程结束，如果t1线程结束或等待超时，则当前线程回到就绪状态。

Thread类提供了多个版本的join()，其定义如下。

- void join()：等待该线程结束。
- void join(long millis)：等待该线程结束的时间最长为millis毫秒。如果超时为0，意味着要一直等下去。
- void join(long millis, int nanos)：等待该线程结束的时间最长为millis毫秒加nanos纳秒。

使用join()方法示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    //共享变量
    static int value = 0;                                      ①

    public static void main(String[ ] args) throws InterruptedException {

        System.out.println("主线程 开始…");

        //创建线程 t1,参数是一个线程执行对象 Runner
        Thread t1 = new Thread(() -> {                         ②
            System.out.println("ThreadA 开始…");
            for (int i = 0; i < 2; i++) {
                System.out.println("ThreadA 执行…");
                value++;                                       ③
            }
            System.out.println("ThreadA 结束…");

        }, "ThreadA");
        //开始线程 t1
        t1.start();
        //主线程被阻塞,等待 t1 线程结束
        t1.join();                                             ④
        System.out.println("value = " + value);                ⑤
        System.out.println("主线程 结束…");
    }
}
```

运行结果如下：

```
主线程 开始…
ThreadA 开始…
ThreadA 执行…
ThreadA 执行…
ThreadA 结束…
value = 2
主线程 结束…
```

上述代码第①行声明了一个共享变量 value，这个变量在子线程中修改，然后主线程访问它。代码第②行采用 Lambda 表达式创建线程，指定线程名为 ThreadA。代码第③行是在子线程 ThreadA 中修改共享变量 value。

代码第④行是在当前线程（主线程）中调用 t1 的 join()方法，因此会导致主线程阻塞，等待 t1 线程结束，从运行结果可以看出主线程被阻塞了。代码第⑤行是打印共享变量 value，从运行结果可见 value ＝ 2。

如果尝试将 t1.join()语句注释掉，输出结果如下：

```
主线程 开始…
value = 0
主线程 结束…
ThreadA 开始…
ThreadA 执行…
ThreadA 执行…
ThreadA 结束…
```

提示 使用 join()方法的场景是，一个线程依赖于另外一个线程的运行结果，所以调用另一个线程的 join()方法等它运行完成。

21.4.3 线程让步

线程类 Thread 还提供一个静态方法 yield()，调用 yield()方法能够使当前线程给其他线程让步。它类似于 sleep()方法，能够使运行状态的线程放弃 CPU 使用权，暂停片刻，然后重新回到就绪状态。与 sleep()方法不同的是，sleep()方法是线程进行休眠，能够给其他线程运行的机会，无论线程优先级高低都有机会运行。而 yield()方法只给相同优先级或更高优先级线程机会。

示例代码如下：

```
//Runner.java 文件
package com.a51work6;

//线程执行对象
public class Runner implements Runnable {

    //编写执行线程代码
    @Override
    public void run() {
```

```
            for (int i = 0; i < 10; i++) {
                //打印次数和线程的名字
                System.out.printf("第 %d 次执行 - %s\n", i,
                            Thread.currentThread().getName());
                Thread.yield();                                    ①
            }
            //线程执行结束
            System.out.println("执行完成! " + Thread.currentThread().getName());
        }
    }
```

代码第①行 Thread.yield()能够使当前线程让步。

提示 yield()方法只能给相同优先级或更高优先级的线程让步，yield()方法在实际开发中很少使用，大多都使用 sleep()方法，sleep()方法可以控制时间，而 yield()方法不能。

21.4.4　线程停止

线程体中的 run()方法结束，线程进入死亡状态，线程就停止了。但是有些业务比较复杂，例如想开发一个下载程序，每隔一段执行一次下载任务，下载任务一般会由子线程执行，休眠一段时间再执行。这个下载子线程中会有一个死循环，为了能够停止子线程，设置一个结束变量。

示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HelloWorld {

    private static String command = "";                           ①

    public static void main(String[] args) {

        //创建线程 t1,参数是一个线程执行对象 Runner
        Thread t1 = new Thread(() -> {

            //一直循环,直到满足条件再停止线程
            while (!command.equalsIgnoreCase("exit")) {           ②
                //线程开始工作
                //TODO
                System.out.println("下载中…");
                try {
                    //线程休眠
```

```
                        Thread.sleep(10000);
                    } catch (InterruptedException e) {
                    }
                }
                //线程执行结束
                System.out.println("执行完成!");
            });
            //开始线程 t1
            t1.start();

            try (InputStreamReader ir = new InputStreamReader(System.in);③
                    BufferedReader in = new BufferedReader(ir)) {
                //从键盘接收了一个字符串的输入
                command = in.readLine(); ④
            } catch (IOException e) {
            }

        }
    }
```

上述代码第①行是设置一个结束变量。代码第②行是在子线程的线程体中判断用户输入的是否为 exit 字符串，如果不是则进行循环，否则结束循环，结束循环就结束了 run()方法，线程就停止了。

代码第③行中的 System.in 是一个很特殊的输入流，能够从控制台（键盘）读取字符。代码第④行是通过流 System.in 读取键盘输入的字符串。测试时需要注意：在控制台输入 exit，然后按 Enter 键，如图 21-6 所示。



图 21-6　在控制台输入字符

提示　控制线程的停止有人会想到使用 Thread 提供的 stop()方法，这个方法已经不推荐使用，因为有时会引发严重的系统故障，类似还有 suspend()和 resume()挂起方法。Java 现在推荐的做法就是采用本例的结束变量方式。

21.5　线程安全

在多线程环境下，访问相同的资源，有可能会引发线程不安全问题。本节讨论引发这些问题的根源和解决方法。

21.5.1　临界资源问题

多一个线程同时运行,有时线程之间需要共享数据,一个线程需要其他线程的数据,否则就不能保证程序运行结果的正确性。

例如有一个航空公司的机票销售,每一天机票数量是有限的,很多售票点同时销售这些机票。下面是一个模拟销售机票系统。示例代码如下:

```
//TicketDB.java 文件
package com.a51work6;

//机票数据库
public class TicketDB {

    //机票的数量
    private int ticketCount = 5;                                ①

    //获得当前机票数量
    public int getTicketCount() {                               ②
        return ticketCount;
    }

    //销售机票
    public void sellTicket() {                                  ③
        try {
            //等于用户付款
            //线程休眠,阻塞当前线程,模拟等待用户付款
            Thread.sleep(1000);                                 ④
        } catch (InterruptedException e) {
        }
        System.out.printf("第%d 号票,已经售出\n", ticketCount);
        ticketCount--;                                          ⑤
    }
}
```

上述代码模拟机票销售过程,代码第①行是声明机票数量成员变量 ticketCount,这是模拟当天可供销售的机票数,为了测试方便初始值设置为 5。代码第②行定义了获取当前机票数的 getTicketCount()方法。代码第③行是销售机票方法,售票网点查询出有没有票可以销售,那么会调用 sellTicket()方法销售机票,这个过程中需要等待用户付款,付款成功后,会将机票数减 1,见代码第⑤行。为模拟等待用户付款,在代码第④行使用了 sleep()方法让当前线程阻塞。

调用代码如下:

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {
```

```
    public static void main(String[] args) {

        TicketDB db = new TicketDB();

        //创建线程 t1
        Thread t1 = new Thread(() -> {
            while (true) {
                int currTicketCount = db.getTicketCount();       ①
                //查询是否有票
                if (currTicketCount > 0) {                       ②
                    db.sellTicket();                             ③
                } else {
                    //无票退出
                    break;
                }
            }
        });
        //开始线程 t1
        t1.start();

        //创建线程 t2
        Thread t2 = new Thread(() -> {
            while (true) {
                int currTicketCount = db.getTicketCount();
                //查询是否有票
                if (currTicketCount > 0) {
                    db.sellTicket();
                } else {
                    //无票退出
                    break;
                }
            }
        });
        //开始线程 t2
        t2.start();
    }
}
```

在 HelloWorld 中创建了两个线程，模拟两个售票网点。首先获得当前机票数量（见代码第①行），然后判断机票数量是否大于 0（见代码第②行），如果有票则出票（见代码第③行），否则退出循环，结束线程。

一次运行结果如下：

```
第 5 号票,已经售出
第 5 号票,已经售出
第 3 号票,已经售出
第 3 号票,已经售出
第 1 号票,已经售出
第 0 号票,已经售出
```

虽然可能每次运行的结果都不一样，但是从结果看还是能发现一些问题：同一张票重复销售、出现第0号票、5张票卖了6次。这些问题的根本原因是多个线程间共享的数据导致数据的不一致性。

提示　多个线程间共享的数据称为共享资源或临界资源，由于是CPU负责线程的调度，程序员无法精确控制多线程的交替顺序。这种情况下，多线程对临界资源的访问有时会导致数据的不一致性。

21.5.2　多线程同步

为了防止多线程对临界资源的访问有时会导致数据的不一致性，Java提供了“互斥”机制，可以为这些资源对象加上一把“互斥锁”，在任一时刻只能由一个线程访问，即使该线程出现阻塞，该对象的被锁定状态也不会解除，其他线程仍不能访问该对象，这就是多线程同步。线程同步是保证线程安全的重要手段，但是线程同步客观上会导致性能下降。

可以通过两种方式实现线程同步，两种方式都涉及synchronized关键字，一种是synchronized方法，使用synchronized关键字修饰方法，对方法进行同步；另一种是synchronized语句，将synchronized关键字放在对象前面限制一段代码的执行。

1. synchronized 方法

synchronized关键字修饰方法实现线程同步，方法所在的对象被锁定，修改21.5.1节售票系统示例。TicketDB.java文件代码如下：

```
//TicketDB.java 文件
package com.a51work6.method;

//机票数据库
public class TicketDB {

    //机票的数量
    private int ticketCount = 5;

    //获得当前机票数量
    public synchronized int getTicketCount() {                ①
        return ticketCount;
    }

    //销售机票
    public synchronized void sellTicket() {                   ②
        try {
            //等于用户付款
            //线程休眠,阻塞当前线程,模拟等待用户付款
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        System.out.printf("第%d号票,已经售出\n", ticketCount);
        ticketCount--;
    }
}
```

上述代码第①行和第②行的方法前都使用了 synchronized 关键字，表明这两个方法是同步的，被锁定的，每一个时刻只能由一个线程访问。并不是每一个方法都有必要加锁的，要仔细研究加锁的必要性，上述代码第①行加锁可以防止出现第 0 号票情况和 5 张票卖出 6 次的情况；代码第②行加锁是防止销售两种一样的票，读者可以自己测试一下。

采用 synchronized 方法修改示例，调用代码 HelloWorld.java 不需要任何修改。

2. synchronized 语句

synchronized 语句方式主要用于第三方类，不方便修改它的代码情况。同样是 21.5.1 节售票系统示例，可以不用修改 TicketDB.java 类，只修改调用代码 HelloWorld.java 实现同步。

HelloWorld.java 代码如下：

```
//HelloWorld.java 文件
package com.a51work6.statement;

public class HelloWorld {

    public static void main(String[ ] args) {

        TicketDB db = new TicketDB();

        //创建线程 t1
        Thread t1 = new Thread(() -> {
            while (true) {
                synchronized(db) {              ①
                    int currTicketCount = db.getTicketCount();
                    //查询是否有票
                    if (currTicketCount > 0) {
                        db.sellTicket();
                    } else {
                        //无票退出
                        break;
                    }
                }
            }
        });
        //开始线程 t1
        t1.start();

        //创建线程 t2
        Thread t2 = new Thread(() -> {
            while (true) {
                synchronized(db) {              ②
                    int currTicketCount = db.getTicketCount();
                    //查询是否有票
                    if (currTicketCount > 0) {
                        db.sellTicket();
                    } else {
```

```
                        //无票退出
                        break;
                    }
                }
            }
        });
        //开始线程 t2
        t2.start();
    }
}
```

代码第①行和第②行是使用 synchronized 语句,将需要同步的代码用大括号括起来。synchronized 后有小括号,将需要同步的对象括起来。

21.6　线程间通信

21.5 节的示例只是简单地为特定对象或方法加锁,但有时情况会更加复杂。如果两个线程之间有依赖关系,线程之间必须进行通信,互相协调才能完成工作。

例如有一个经典的堆栈问题,一个线程生成了一些数据,将数据压栈;另一个线程消费了这些数据,将数据出栈。这两个线程互相依赖,当堆栈为空,消费线程无法取出数据时,应该通知生成线程添加数据;当堆栈已满,生产线程无法添加数据时,应该通知消费线程取出数据。

为了实现线程间通信,需要使用 Object 类中声明的 5 个方法。

- void wait():使当前线程释放对象锁,然后当前线程处于对象等待队列中阻塞状态,如图 21-7 所示,等待其他线程唤醒。

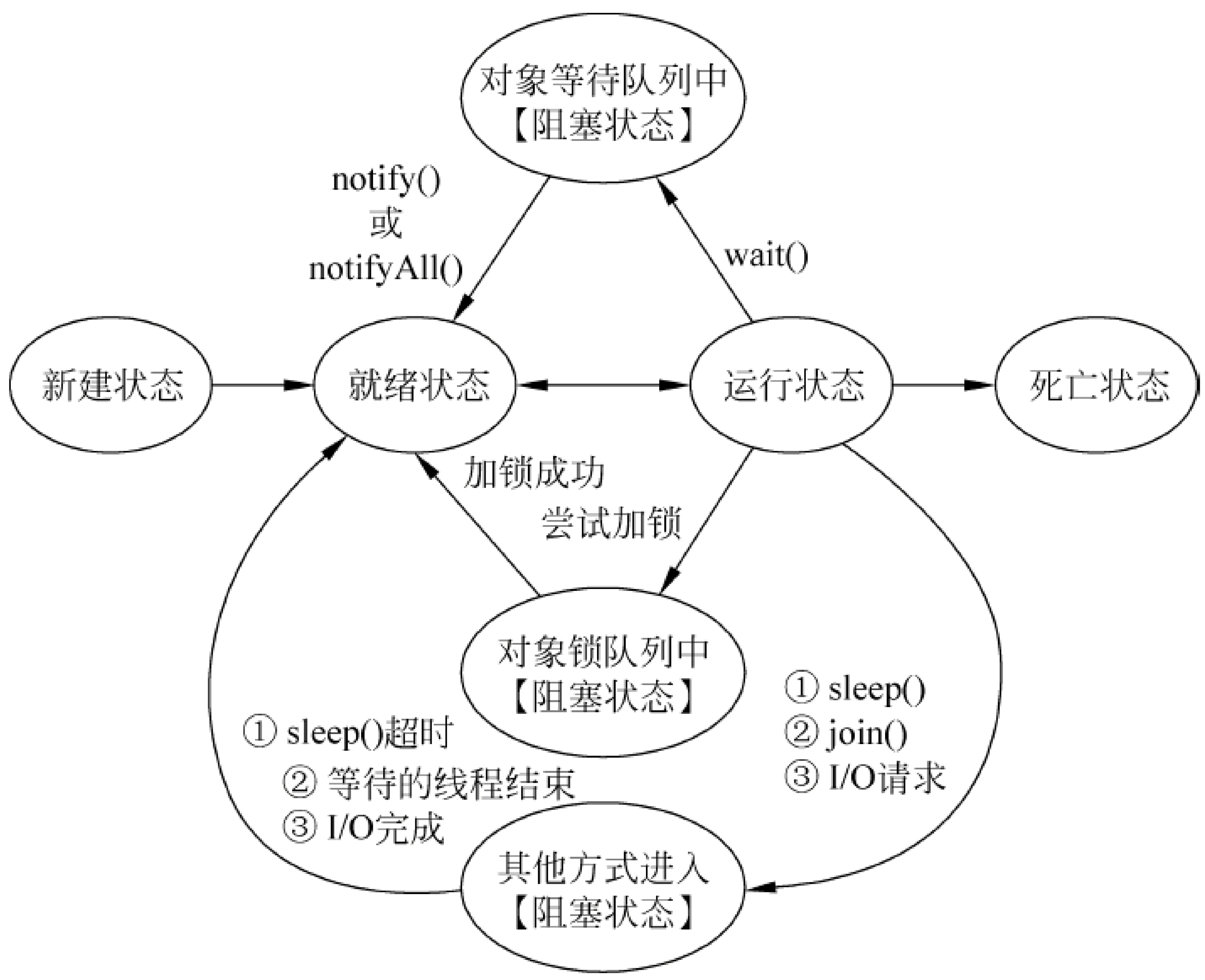


图 21-7　线程间通信

- void wait(long timeout)：同 wait()方法，等待 timeout 毫秒时间。
- void wait(long timeout, int nanos)：同 wait()方法，等待 timeout 毫秒加 nanos 纳秒时间。
- void notify()：当前线程唤醒此对象等待队列中的一个线程，如图 21-7 所示，该线程将进入就绪状态。
- void notifyAll()：当前线程唤醒此对象等待队列中的所有线程，如图 21-7 所示，这些线程将进入就绪状态。

提示 图 21-7 是图 21-5 的补充，从图 21-7 可见，线程有多种方式进入阻塞状态，除了通过 wait() 外，还有加锁的方式和其他方式，加锁方式是 21.5 节介绍的使用 synchronized 加互斥锁；其他方式事实上是 21.3 节介绍的方式，这里不再赘述。

下面看看消费和生产示例中堆栈类代码：

```
//Stack.java 文件
package com.a51work6;

//堆栈类
class Stack {
    //堆栈指针初始值为 0
    private int pointer = 0;
    //堆栈有 5 个字符的空间
    private char[] data = new char[5];

    //压栈方法,加上互斥锁
    public synchronized void push(char c) {                ①
        //堆栈已满,不能压栈
        while (pointer == data.length) {
            try {
                //等待,直到有数据出栈
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        //通知其他线程把数据出栈
        this.notify();
        //数据压栈
        data[pointer] = c;
        //指针向上移动
        pointer++;
    }

    //出栈方法,加上互斥锁
    public synchronized char pop() {                       ②
        //堆栈无数据,不能出栈
        while (pointer == 0) {
            try {
                //等待其他线程把数据压栈
                this.wait();
```

```
                } catch (InterruptedException e) {
                }
            }
            //通知其他线程压栈
            this.notify();
            //指针向下移动
            pointer--;
            //数据出栈
            return data[pointer];
        }
    }
```

上述代码实现了同步堆栈类，该堆栈有最多5个元素的空间，代码第①行声明了压栈方法push()，该方法是一个同步方法，在该方法中首先判断是否堆栈已满，如果已满，则不能压栈，调用this.wait()让当前线程进入对象等待状态中。如果堆栈未满，则程序会往下运行调用this.notify()唤醒对象等待队列中的一个线程。代码第②行声明了出栈方法pop()方法，与push()方法类似，这里不再赘述。

调用代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

public class HelloWorld {

    public static void main(String args[]) {

        Stack stack = new Stack();                              ①

        //下面的消费者和生产者所操作的是同一个堆栈对象 stack
        //生产者线程
        Thread producer = new Thread(() -> {                    ②
            char c;
            for (int i = 0; i < 10; i++) {
                //随机产生 10 个字符
                c = (char) (Math.random() * 26 + 'A');
                //把字符压栈
                stack.push(c);
                //打印字符
                System.out.println("生产: " + c);
                try {
                    //每产生一个字符线程就睡眠
                    Thread.sleep((int) (Math.random() * 1000));
                } catch (InterruptedException e) {
                }
            }

        });
```

```
        //消费者线程
        Thread consumer = new Thread(() -> {③
            char c;
            for (int i = 0; i < 10; i++) {
                //从堆栈中读取字符
                c = stack.pop();
                //打印字符
                System.out.println("消费: " + c);
                try {
                    //每读取一个字符线程就睡眠
                    Thread.sleep((int) (Math.random() * 1000));
                } catch (InterruptedException e) {
                }
            }

        });

        producer.start();              //启动生产者线程
        consumer.start();              //启动消费者线程
    }
}
```

上述代码第①行创建堆栈对象,代码第②行创建生产者线程,代码第③行创建消费者线程。

本章小结

本章介绍了Java线程技术。首先介绍了线程相关的概念,然后介绍了如何创建子线程、线程状态、线程管理、线程安全和线程间通信等内容,其中创建线程和线程管理是学习的重点,应掌握线程状态和线程安全,了解线程间通信。

21.7 同步练习

1. 下面哪些方法可以线程放弃CPU使用权?( )
   A. sleep()   B. wait()
   C. notifyAll()   D. yield()
2. 下列哪个选项可用于创建一个可运行的类?( )
   A. public class X implements Runnable{ public void run(){…} }
   B. public class X implements Thread{ public void run(){…} }
   C. public class X implements Thread{ public int run(){…} }
   D. public class X implements Runnable{ protected void run(){…} }
3. 运行下列程序,会产生什么结果?( )

```
public class X extends Thread implements Runnable {
    public void run() {
```

```
        System.out.println("this is run()");
    }

    public static void main(String args[]) {
        Thread t = new Thread(new X());
        t.start();
    }
}
```

A. 第 1 行会产生编译错误　　B. 第 6 行会产生编译错误

C. 第 6 行会产生运行错误　　D. 程序会运行和启动

4. 哪个关键字可以对对象加互斥锁？(　　)

A. transient　　B. synchronized

C. serialize　　D. static

第 22 章 网络编程

CHAPTER 22

现在的应用程序都离不开网络,网络编程是非常重要的技术。Java SE 提供了 java.net 包,其中包含了网络编程所需要的最基础一些类和接口。这些类和接口面向两个不同的层次:基于 Socket 的低层次网络编程和基于 URL 的高层次网络编程。所谓高低层次,就是通信协议的高低层次,Socket 采用 TCP、UDP 等协议,这些协议属于低层次的通信协议;URL 采用 HTTP 和 HTTPS,这些属于高层次的通信协议。低层次网络编程,因为它面向底层,比较复杂,但是"低层次网络编程"并不等于它功能不强大。恰恰相反,正因为层次低,Socket 编程与基于 URL 的高层次网络编程相比,能够提供更强大的功能和更灵活的控制,但是要更复杂一些。

本章会介绍基于 Socket 的低层次网络编程和基于 URL 的高层次网络编程,以及数据交换格式。

22.1 网络基础

网络编程需要程序员掌握一些基础的网络知识,本节介绍一些网络基础知识。

22.1.1 网络结构

首先了解一下网络结构。网络结构是网络的构建方式,目前流行的有客户端/服务器结构网络和对等结构网络。

1. 客户端/服务器结构网络

客户端/服务器(Client Server,C/S)结构网络是一种主从结构网络。如图 22-1 所示,服务器一般处于等待状态,如果有客户端请求,服务器响应请求建立连接,提供服务。服务器是被动的,有点像在餐厅吃饭时的服务员。而客户端是主动的,像在餐厅吃饭的顾客。

事实上,生活中很多网络服务都采用这种结构,如 Web 服务、文件传输服务和邮件服务等。虽然它们存在的目的不一样,但基本结构是一样的。这种网络结构与设备类型无关,服务器不一定是计算机,也可能是手机等移动设备。

2. 对等结构网络

对等结构网络也叫点对点网络(Peer to Peer,P2P),每个节点之间是对等的。如图 22-2 所示,每个节点既是服务器又是客户端,这种结构有点像吃自助餐。

对等结构网络分布范围比较小。通常在一间办公室或一个家庭内,因此它非常适合于

移动设备间的网络通信，网络链路层由蓝牙和 WiFi 实现。

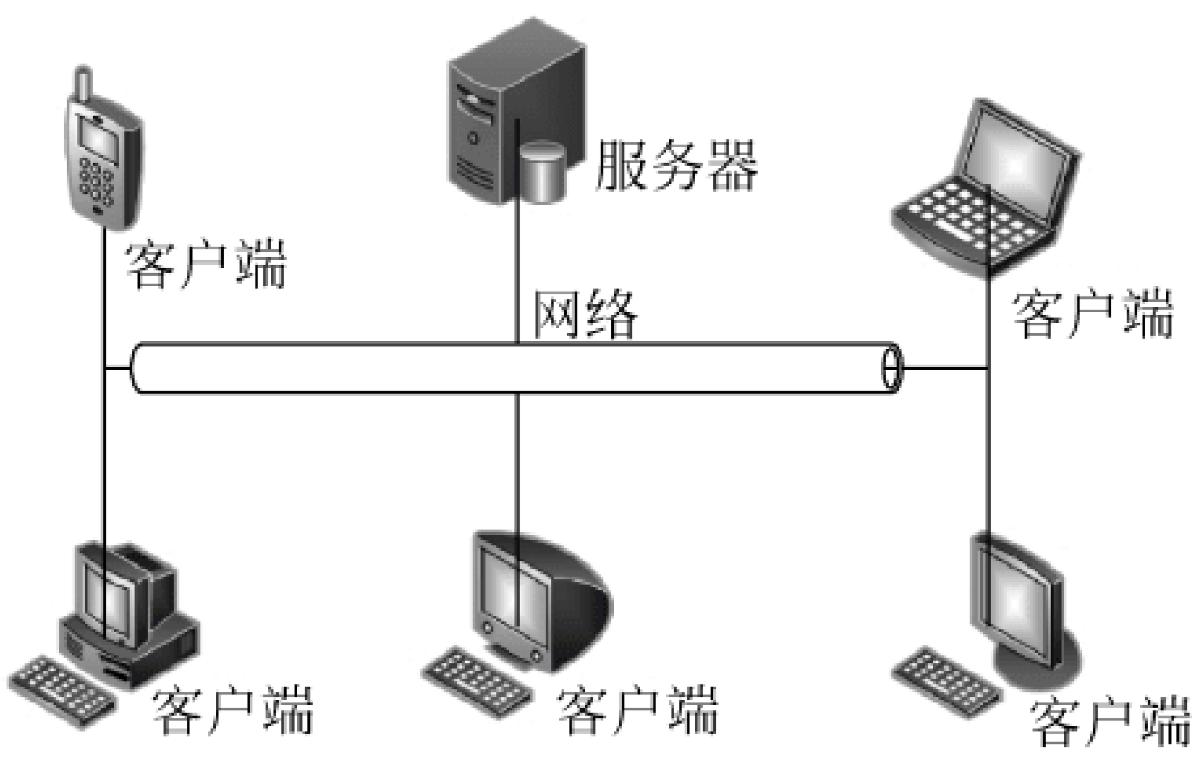


图 22-1　客户端/服务器结构网络

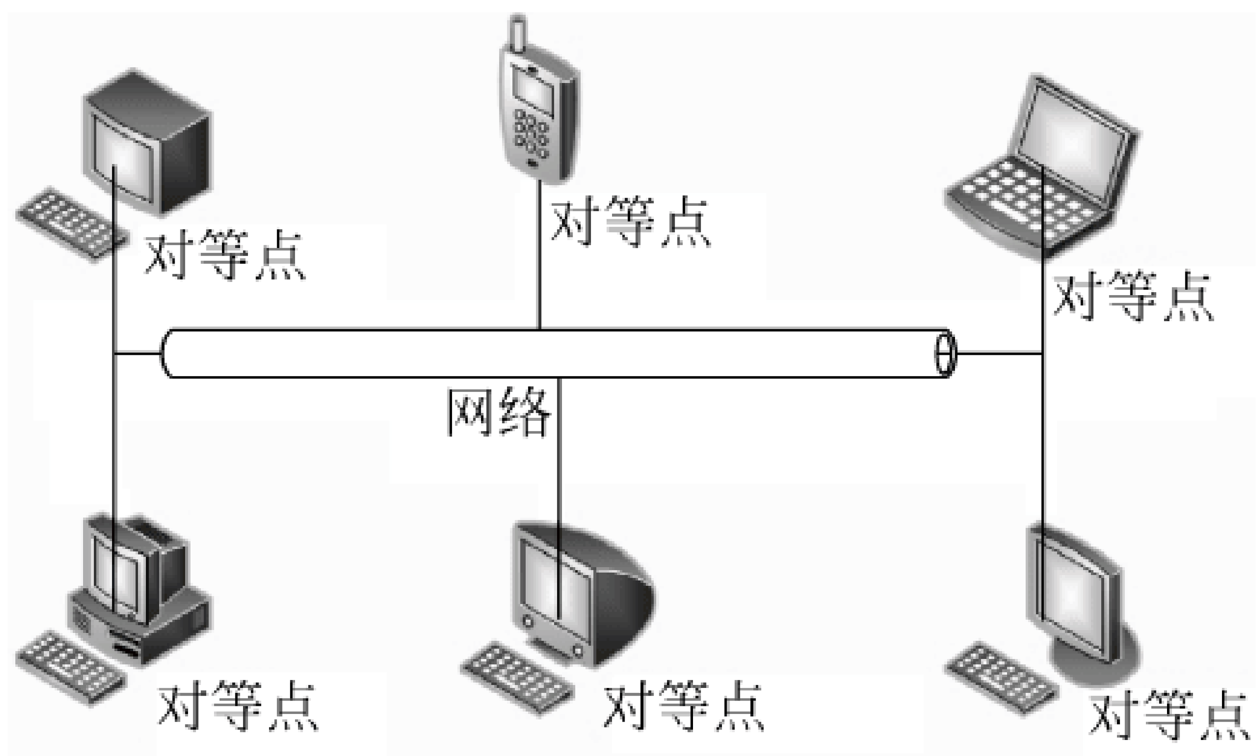


图 22-2　对等结构网络

22.1.2　TCP/IP 协议

网络通信会用到协议，其中 TCP/IP 协议是非常重要的。TCP/IP 协议是由 IP 和 TCP 两个协议构成的。IP(Internet Protocol)协议是一种低级的路由协议，它将数据拆分成许多小的数据包，并通过网络将它们发送到某一特定地址，但无法保证所有包都抵达目的地，也不能保证包的顺序。

由于 IP 协议传输数据的不安全性，网络通信时还需要 TCP 协议。传输控制协议(Transmission Control Protocol，TCP)是一种高层次的协议，是面向连接的可靠数据传输协议，如果有些数据包没有收到，则会重发，并对数据包内容准确性检查，而且保证数据包顺序，所以该协议保证数据包能够安全地按照发送顺序送达目的地。

22.1.3　IP 地址

为实现网络中不同计算机之间的通信，每台计算机都必须有一个与众不同的标识，这就是 IP 地址，TCP/IP 使用 IP 地址来标识源地址和目的地址。最初所有的 IP 地址都是 32 位数字，由 4 个 8 位的二进制数组成，每 8 位之间用圆点隔开，如 192.168.1.1，这种类型的地址通过 IPv4 指定。而现在有一种新的地址模式称为 IPv6，IPv6 使用 128 位数字表示一个地址，分为 8 个 16 位块。尽管 IPv6 比 IPv4 有很多优势，但是由于习惯的问题，很多设备还是采用 IPv4。不过 Java 语言同时采用 IPv4 和 IPv6。

在 IPv4 地址模式中，IP 地址分为 A、B、C、D 和 E 等 5 类。

- A 类地址用于大型网络，地址为：1.0.0.1～126.155.255.254。
- B 类地址用于中型网络，地址为：128.0.0.1～191.255.255.254。
- C 类地址用于小规模网络，地址为：192.0.0.1～223.255.255.254。
- D 类地址用于多目的地信息的传输和作为备用。
- E 类地址保留，仅做实验和开发用。

另外，有时还会用到一个特殊的 IP 地址 127.0.0.1，127.0.0.1 称为回送地址，指本机。主要用于网络软件测试以及本地机进程间通信，使用回送地址发送数据，不进行任何网络传输，只在本机进程间通信。

22.1.4 端口

一个 IP 地址标识一台计算机,每一台计算机又有很多网络通信程序在运行,提供网络服务或进行通信,这就需要不同的端口进行通信。如果把 IP 地址比作电话号码,那么端口就是分机号码,进行网络通信时不仅要指定 IP 地址,还要指定端口号。

TCP/IP 系统中的端口号是一个 16 位的数字,它的端口号是 0～65535。小于 1024 的端口号保留给预定义的服务,如 HTTP 是 80,FTP 是 21,Telnet 是 23,E-mail 是 25 等。除非要和那些服务进行通信,否则不应该使用小于 1024 的端口。

22.2 TCP Socket 低层次网络编程

TCP/IP 协议的传输层有两种传输协议:TCP(传输控制协议)和 UDP(用户数据包协议)。TCP 是面向连接的可靠数据传输协议。TCP 就好比电话,电话接通后双方才能通话,在挂断电话之前,电话一直占线。TCP 连接一旦建立起来,一直占用,直到关闭连接。另外,TCP 为了保证数据的正确性,会重发一切没有收到的数据,还会对数据内容进行验证,并保证数据传输的正确顺序。因此 TCP 协议对系统资源的要求较多。

基于 TCP Socket 编程很有代表性,下面介绍 TCP Socket 编程。

22.2.1 TCP Socket 通信简介

Socket 是网络上的两个程序,通过一个双向的通信连接,实现数据的交换。这个双向链路的一端称为一个 Socket。Socket 通常用来实现客户端和服务器端的连接。Socket 是 TCP/IP 协议的一个十分流行的编程接口,一个 Socket 由一个 IP 地址和一个端口号唯一确定,一旦建立连接,Socket 还会包含本机和远程主机的 IP 地址和端口号,如图 22-3 所示,Socket 是成对出现的。

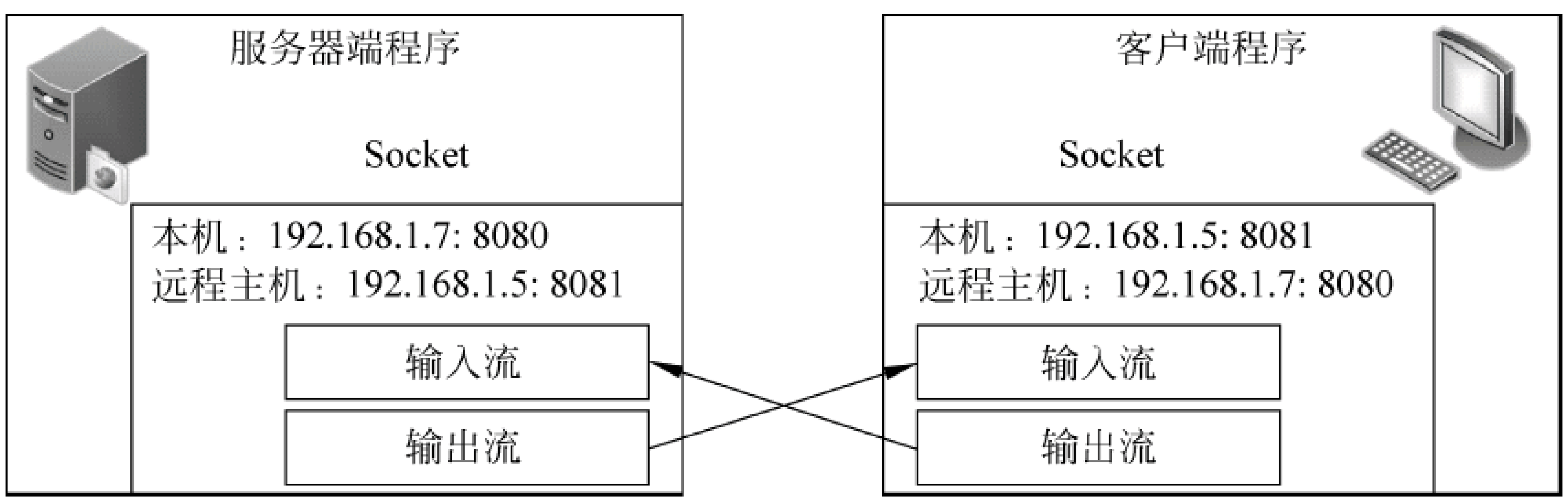


图 22-3 TCP Socket 通信

22.2.2 TCP Socket 通信过程

使用 Socket 进行 C/S 结构编程,通信过程如图 22-4 所示。

服务器端监听某个端口是否有连接请求。服务器端程序处于阻塞状态,直到客户端向服务器端发出连接请求。服务器端接收客户端请求。服务器端会响应请求,处理请求,然后将结果应答给客户端,这样就会建立连接。一旦连接建立起来,通过 Socket 可以获得 I/O

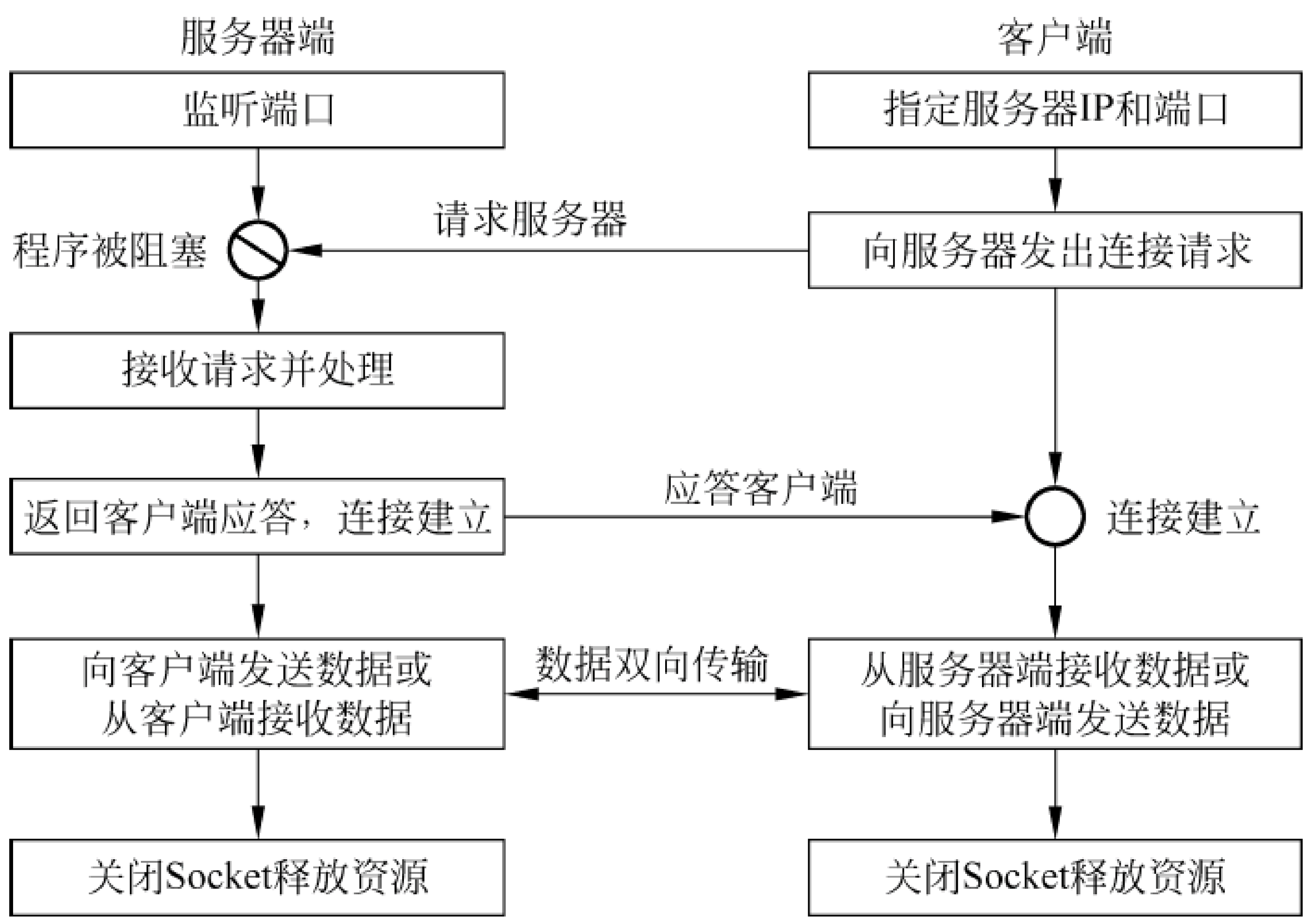


图 22-4　TCP Socket 通信过程

流对象。借助于 I/O 流对象就可以实现服务器端与客户端的通信，最后不要忘记关闭 Socket 和释放一些资源(包括关闭 I/O 流)。

22.2.3　Socket 类

java.net 包为 TCP Socket 编程提供了两个核心类：Socket 和 ServerSocket，分别用来表示双向连接的客户端和服务器端。

本节先介绍一下 Socket 类。Socket 常用的构造方法如下。

- Socket(InetAddress address, int port)：创建 Socket 对象，并指定远程主机 IP 地址和端口号。
- Socket(InetAddress address, int port, InetAddress localAddr, int localPort)：创建 Socket 对象，并指定远程主机 IP 地址和端口号，以及本机的 IP 地址(localAddr)和端口号(localPort)。
- Socket(String host, int port)：创建 Socket 对象，并指定远程主机名和端口号，IP 地址为 null，null 表示回送地址，即 127.0.0.1。
- Socket(String host, int port, InetAddress localAddr, int localPort)：创建 Socket 对象，并指定远程主机和端口号，以及本机的 IP 地址(localAddr)和端口号(localPort)。host 为主机名，IP 地址为 null，null 表示回送地址，即 127.0.0.1。

Socket 其他的常用方法如下。

- InputStream getInputStream()：通过此 Socket 返回输入流对象。
- OutputStream getOutputStream()：通过此 Socket 返回输出流对象。
- int getPort()：返回 Socket 连接到的远程端口。
- int getLocalPort()：返回 Socket 绑定到的本地端口。
- InetAddress getInetAddress()：返回 Socket 连接的地址。
- InetAddress getLocalAddress()：返回 Socket 绑定的本地地址。
- boolean isClosed()：返回 Socket 是否处于关闭状态。

- boolean isConnected()：返回 Socket 是否处于连接状态。
- void close()：关闭 Socket。

22.2.4 ServerSocket 类

ServerSocket 常用的构造方法如下。

- ServerSocket(int port, int maxQueue)：创建绑定到特定端口的服务器 Socket。maxQueue 设置连接的请求最大队列长度，如果队列满，则拒绝该连接。默认值是 50。
- ServerSocket(int port)：创建绑定到特定端口的服务器 Socket。最大队列长度是 50。

ServerSocket 其他的常用方法如下。

- InputStream getInputStream()：通过此 Socket 返回输入流对象。
- OutputStream getOutputStream()：通过此 Socket 返回输出流对象。
- boolean isClosed()：返回 Socket 是否处于关闭状态。
- Socket accept()：侦听并接收到 Socket 的连接。此方法在建立连接之前一直阻塞。
- void close()：关闭 Socket。

ServerSocket 类本身不能直接获得 I/O 流对象，而是通过 accept()方法返回 Socket 对象，通过 Socket 对象取得 I/O 流对象并进行网络通信。此外，ServerSocket 也实现了 AutoCloseable 接口，通过自动资源管理技术关闭 ServerSocket。

注意 Socket 与流类似，所占用的资源不能通过 JVM 的垃圾收集器回收，需要程序员释放。一种方法是可以在 finally 代码块调用 close()方法关闭 Socket，释放流所占用的资源；另一种方法是通过自动资源管理技术释放资源，Socket 和 ServerSocket 都实现了 AutoCloseable 接口。

22.2.5 案例：文件上传工具

基于 TCP Socket 编程比较复杂，这里先从一个简单的文件上传工具案例介绍 TCP Socket 编程基本流程。上传过程是一个单向 Socket 通信过程，如图 22-5 所示，客户端通过文件输入流读取文件，然后从 Socket 获得输出流写入数据，写入数据完成上传成功，客户端任务完成。服务器端先从 Socket 获得输入流，然后写入文件输出流，写入数据完成上传成功，服务器端任务完成。

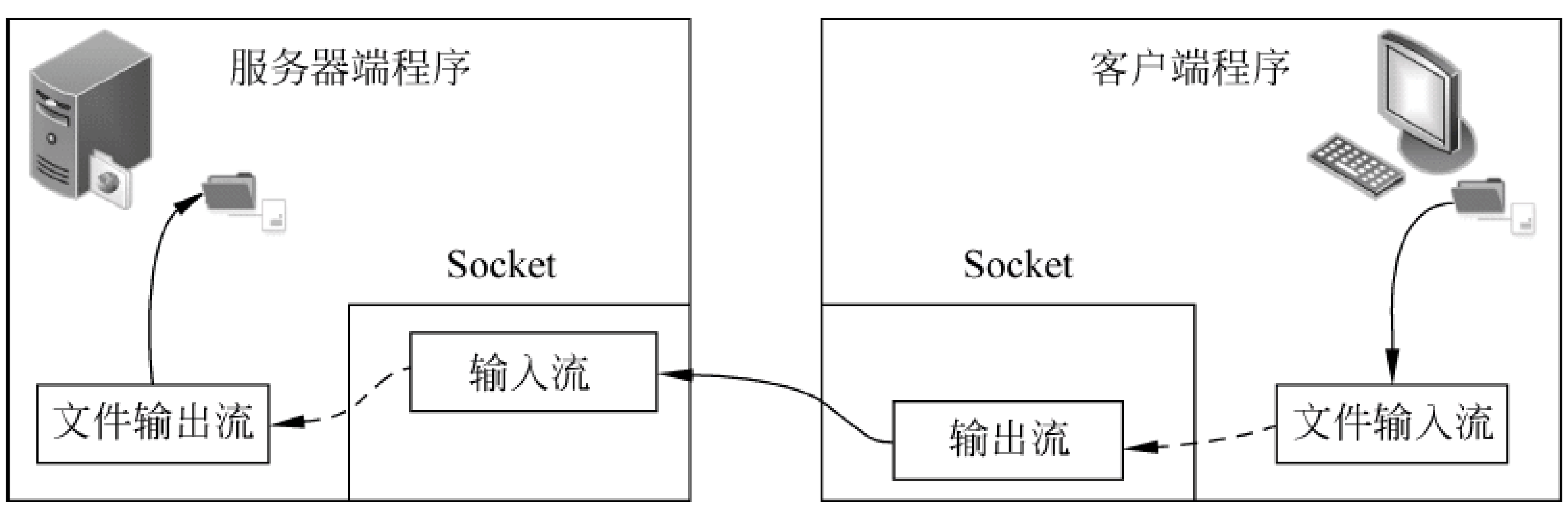


图 22-5 单向 Socket 通信

服务器端 UploadServer 代码如下：

```
//UploadServer.java 文件
package com.a51work6;
…
public class UploadServer {

    public static void main(String[ ] args) {

        System.out.println("服务器端运行…");

        try ( //创建一个 ServerSocket 监听 8080 端口的客户端请求
                ServerSocket server = new ServerSocket(8080);           ①
                //使用 accept()阻塞当前线程,等待客户端请求
                Socket socket = server.accept();                        ②
                //由 Socket 获得输入流,并创建缓冲输入流
                BufferedInputStream in = new BufferedInputStream(socket
.getInputStream());                                                     ③
                //由文件输出流创建缓冲输出流
                 FileOutputStream out = new FileOutputStream("./TestDir/subDir/coco2dxcplus.
jpg")) {                                                                ④

              //准备一个缓冲区
              byte[ ] buffer = new byte[1024];
              //首次从 Socket 读取数据
              int len = in.read(buffer);
              while (len != -1) {
                  //写入数据到文件
                  out.write(buffer, 0, len);
                  //再次从 Socket 读取数据
                  len = in.read(buffer);
              }

              System.out.println("接收完成!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

上述代码第①行创建 ServerSocket 对象监听本机的 8080 端口，这时当前线程还没有阻塞，调用代码第②行的 server.accept()才会阻塞当前线程，等待客户端请求。

提示 由于当前线程是主线程，所以 server.accept()会阻塞主线程。阻塞主线程是不明智的，如果是在一个图形界面的应用程序，阻塞主线程会导致无法进行任何的界面操作，就是常见的“卡”现象，所以最好是把 server.accept()语句放到子线程中。

代码第③行是从 socket 对象中获得输入流对象，代码第④行是文件输出流。其后的输入输出代码可以参考第 20 章，这里不再赘述。

客户端UploadClient代码如下：

```
//UploadClient.java 文件
package com.a51work6;
…
public class UploadClient {

    public static void main(String[] args) {

        System.out.println("客户端运行…");

        try ( //向本机的 8080 端口发出请求
                Socket socket = new Socket("127.0.0.1", 8080);              ①
                //由 Socket 获得输出流,并创建缓冲输出流
                BufferedOutputStream out = new BufferedOutputStream (socket.getOutputStream());
                                                                            ②
                //创建文件输入流
                FileInputStream fin = new FileInputStream("./TestDir/coco2dxcplus.jpg");
                //由文件输入流创建缓冲输入流
                BufferedInputStream in = new BufferedInputStream(fin)) {

            //准备一个缓冲区
            byte[] buffer = new byte[1024];
            //首次读取文件
            int len = in.read(buffer);
            while (len != -1) {
                //数据写入 Socket
                out.write(buffer, 0, len);
                //再次读取文件
                len = in.read(buffer);
            }

            System.out.println("上传成功!");

        } catch (ConnectException e) {                                      ③
            System.out.println("服务器未启动!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行创建Socket，指定远程主机的IP地址和端口号。代码第②行是从socket对象获得输出流。代码第③行是捕获ConnectException异常，这个异常引起的原因是在代码第①行向服务器发出请求时，服务器拒绝了客户端请求，这有两种可能性：一是服务器没有启动，服务器的8080端口没有打开；二是服务器请求队列已满(默认是50个)。

提示 案例测试时，先运行服务器，再运行客户端。测试Socket程序最好打开两个命令行窗口，通过java指令分别运行服务器程序和客户端程序，如图22-6和图22-7所示。需要注意当前运行的路径是Eclipse工程根目录，需要指定类路径，命令的-cp .;./bin就是

指定类路径，包括两个当前路径：其中点(.)表示当前路径，./bin 表示 bin 目录，也可以写成.\bin。为什么要指定 bin 目录呢？是因为编译之后的字节码文件放在此目录中。另外，如果想在 Eclipse 中查看多个控制台信息，如图 22-8 所示，在控制台上面的工具栏中单击“选择控制台”按钮可以实现切换。

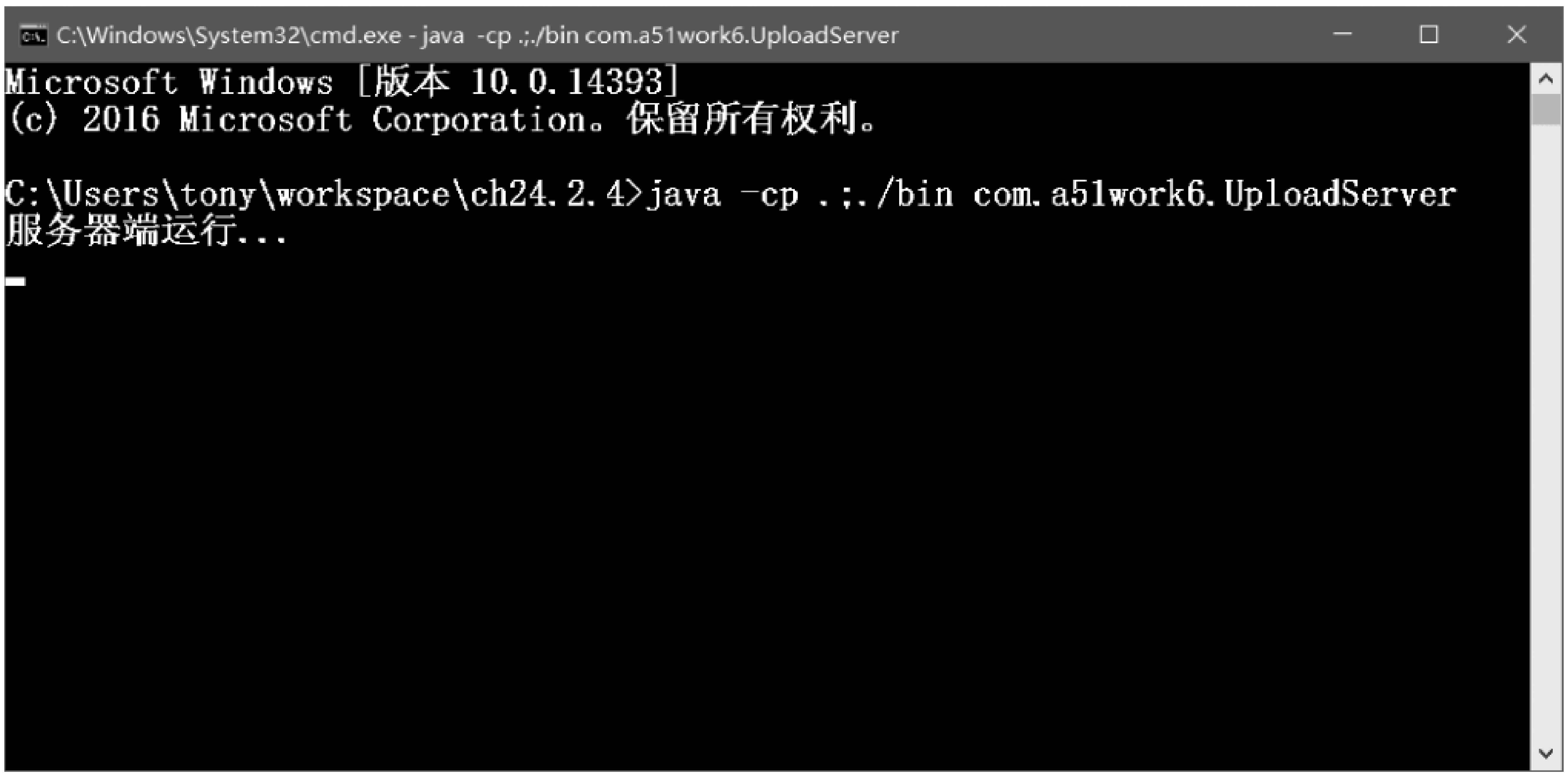


图 22-6 命令行窗口运行服务器程序

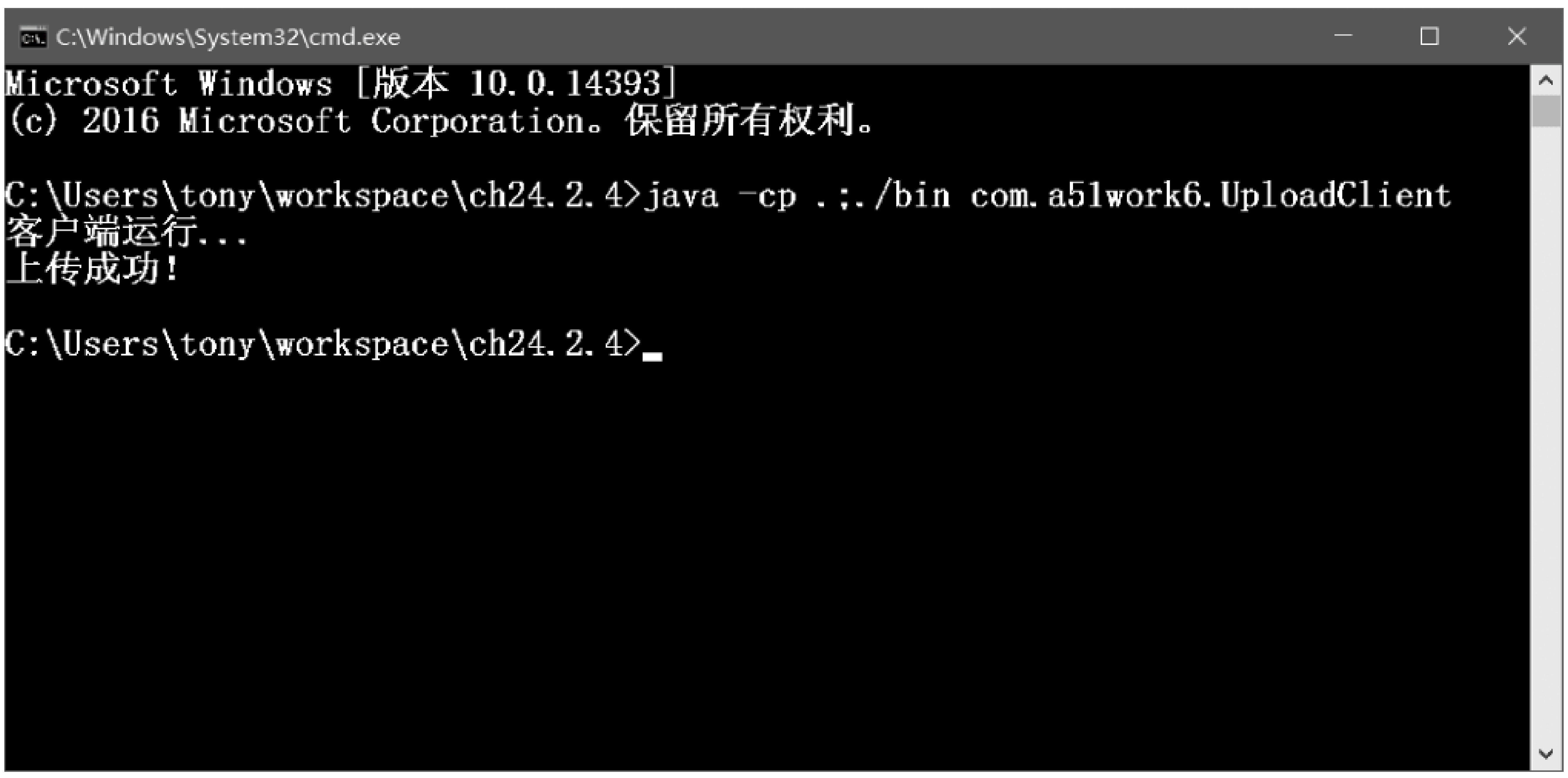


图 22-7 命令行窗口运行客户端程序

22.2.6 案例：聊天工具

22.2.5 节介绍的案例只是单向传输的 Socket，Socket 可以双向数据传输，但是这就有些复杂了，比较有代表性的案例就是聊天工具。

如图 22-9 所示是基于 TCP Socket 聊天工具案例，其中的标准输入是键盘，标准输出是显示器的控制台。首先客户端通过键盘输入字符串，通过标准输入流读取字符串，然后通过

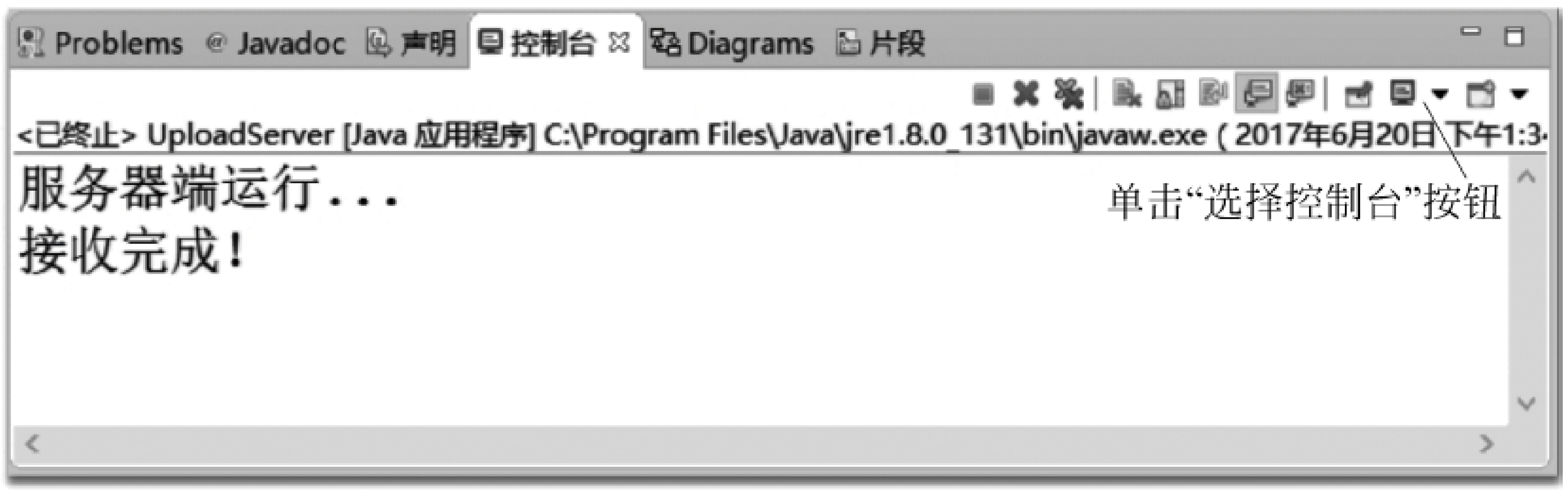


图 22-8　Eclipse 中切换控制台

Socket 获得输出流，将字符串写入输出流。接着服务器通过 Socket 获得输入流，从输入流中读取来自客户端发送过来的字符串，然后通过标准输出流输出到显示器的控制台。服务器向客户端传送字符串过程与此类似。

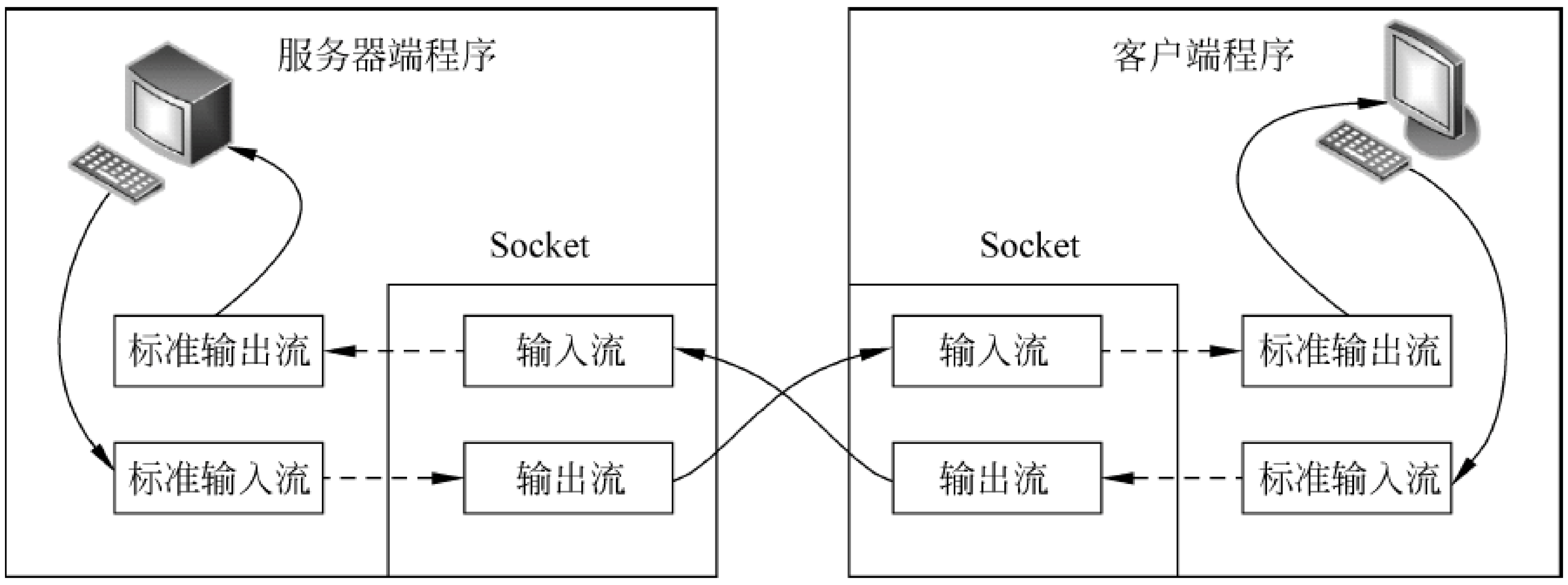


图 22-9　基于 TCP Socket 聊天工具案例

服务器端 ChatServer 代码如下：

```
//ChatServer.java 文件
package com.a51work6;
…
public class ChatServer {

    public static void main(String[] args) {

        System.out.println("服务器运行…");

        Thread t = new Thread(() -> {                                                ①

            try ( //创建一个 ServerSocket 监听端口 8080 客户请求
                    ServerSocket server = new ServerSocket(8080);
                    //使用 accept()阻塞等待客户端请求
                    Socket socket = server.accept();
                    DataInputStream in = new DataInputStream(socket.getInputStream());   ②
                    DataOutputStream out = new DataOutputStream(socket.getOutputStream()); ③
                    BufferedReader keyboardIn
                            = new BufferedReader(new InputStreamReader(System.in))) {    ④
```

```
                while (true) {
                    /* 接收数据 */
                    String str = in.readUTF();                                    ⑤
                    //打印接收的数据
                    System.out.printf("从客户端接收的数据:【%s】\n", str);

                    /* 发送数据 */
                    //读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    //结束聊天
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    //发送
                    out.writeUTF(keyboardInputString);                            ⑥
                    out.flush();
                }
            } catch (Exception e) {
            }
            System.out.println("服务器停止…");
        });

        t.start();
    }
}
```

上述代码第①行是创建一个子线程，将网络通信放到子线程中处理是一种很好的做法，因为网络通信往往有线程阻塞过程，放到子线程中处理就不会阻塞主线程了。

代码第②行是从 Socket 中获得数据输入流，代码第③行是从 Socket 中获得数据输出流，数据流主要面向基本数据类型，本例中使用它们主要用来输入输出 UTF 编码的字符串，代码第⑤行的 readUTF()是数据输入流读取字符串。代码第⑥行的 writeUTF()是数据输出流写入字符串。代码第④行中的 System.in 是标准输入流，然后使用标准输入流创建缓冲输入流。

客户端 ChatClient 代码如下：

```
//ChatClient.java 文件
package com.a51work6;
…
public class ChatClient {

    public static void main(String[] args) {

        System.out.println("客户端运行…");

        Thread t = new Thread(() -> {

            try ( //向 127.0.0.1 主机 8080 端口发出连接请求
                    Socket socket = new Socket("127.0.0.1", 8080);
```

```
                    DataInputStream in = new DataInputStream(socket.getInputStream());
                    DataOutputStream out = new DataOutputStream(socket.getOutputStream());
                    BufferedReader keyboardIn
                        = new BufferedReader(new InputStreamReader(System.in))) {

                while (true) {
                    /* 发送数据 */
                    //读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    //结束聊天
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    //发送
                    out.writeUTF(keyboardInputString);
                    out.flush();

                    /* 接收数据 */
                    String str = in.readUTF();
                    //打印接收的数据
                    System.out.printf("从服务器接收的数据:【%s】\n", str);
                }
            } catch (ConnectException e) {
                System.out.println("服务器未启动!");
            } catch (Exception e) {
            }
            System.out.println("客户端停止!");
        });

        t.start();

    }
}
```

客户端ChatClient代码与服务器端ChatServer代码类似，这里不再赘述。

22.3 UDP Socket低层次网络编程

UDP(用户数据包协议)就像日常生活中的邮件投递，不能保证可靠地寄到目的地的。UDP是无连接的，对系统资源的要求较少，UDP可能丢包，也不保证数据顺序。但是对于网络游戏和在线视频等要求传输快、实时性高、质量可稍差一点儿的数据传输，UDP还是非常不错的。

UDP Socket网络编程比TCP Socket编程简单得多，UDP是无连接协议，不需要像TCP一样监听端口，建立连接，然后才能进行通信。

22.3.1　DatagramSocket 类

java.net 包中提供了两个类：DatagramSocket 和 DatagramPacket，用来支持 UDP 通信。本节介绍 DatagramSocket 类，DatagramSocket 用于在程序之间建立传送数据包的通信连接。

DatagramSocket 常用的构造方法如下。

- DatagramSocket()：创建数据包 DatagramSocket 对象，并将其绑定到本地主机上任何可用的端口。
- DatagramSocket(int port)：创建数据包 DatagramSocket 对象，并将其绑定到本地主机上的指定端口。
- DatagramSocket(int port, InetAddress laddr)：创建数据包 DatagramSocket 对象，并将其绑定到指定的本地地址。

DatagramSocket 其他的常用方法如下。

- void send(DatagramPacket p)：发送数据包。
- void receive(DatagramPacket p)：接收数据包。
- int getPort()：返回 DatagramSocket 连接到的远程端口。
- int getLocalPort()：返回 DatagramSocket 绑定到的本地端口。
- InetAddress getInetAddress()：返回 DatagramSocket 连接的地址。
- InetAddress getLocalAddress()：返回 DatagramSocket 绑定的本地地址。
- boolean isClosed()：返回 DatagramSocket 是否处于关闭状态。
- boolean isConnected()：返回 DatagramSocket 是否处于连接状态。
- void close()：关闭 DatagramSocket。

DatagramSocket 也实现了 AutoCloseable 接口，通过自动资源管理技术关闭 DatagramSocket。

22.3.2　DatagramPacket 类

DatagramPacket 用来表示数据包，是数据传输的载体。DatagramPacket 实现无连接数据包投递服务，每次投递数据包仅根据该包中信息从一台机器路由到另一台机器。从一台机器发送到另一台机器的多个包可能选择不同的路由，也可能按不同的顺序到达，不保证包都能到达目的地。

DatagramPacket 的构造方法如下。

- DatagramPacket(byte[] buf, int length)：构造数据包，buf 是包数据，length 是接收包数据的长度。
- DatagramPacket(byte[] buf, int length, InetAddress address, int port)：构造数据包，包发送到指定主机上的指定端口号。
- DatagramPacket(byte[] buf, int offset, int length)：构造数据包，offset 是 buf 字节数组的偏移量。
- DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)：构造数据包，包发送到指定主机上的指定端口号。

DatagramPacket 常用的方法如下。

- InetAddress getAddress()：返回发往或接收该数据包相关的主机的 IP 地址。
- byte[] getData()：返回数据包中的数据。
- int getLength()：返回发送或接收到的数据(byte[])的长度。
- int getOffset()：返回发送或接收到的数据(byte[])的偏移量。
- int getPort()：返回发往或接收该数据包相关主机的端口号。

22.3.3 案例：文件上传工具

使用 UDP Socket 将 22.2.5 节文件上传工具重新实现一下。

服务器端 UploadServer 代码如下：

```
//UploadServer.java 文件
package com.a51work6;
…
public class UploadServer {
    public static void main(String args[]) {

        System.out.println("服务器端运行…");

        //创建一个子线程
        Thread t = new Thread(() -> {                                              ①

            try ( //创建 DatagramSocket 对象,指定端口 8080
                    DatagramSocket socket = new DatagramSocket(8080);              ②
                    FileOutputStream fout = new FileOutputStream("./TestDir/subDir/coco2dxcplus.
jpg");
                    BufferedOutputStream out = new BufferedOutputStream (fout)) {

                //准备一个缓冲区
                byte[] buffer = new byte[1024];

                //循环接收数据包
                while (true) {

                    //创建数据包对象,用来接收数据
                    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                    //接收数据包
                    socket.receive(packet);
                    //接收数据长度
                    int len = packet.getLength();

                    if (len == 3) {                                                ③
                        //获得结束标志
                        String flag = new String(buffer, 0, 3);
                        //判断结束标志,如果是 bye,则结束接收
                        if (flag.equals("bye")) {                                  ④
                            break;
                        }
                    }
```

```
                    //写入数据到文件输出流
                    out.write(buffer, 0, len);
                }
                System.out.println("接收完成!");
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        //启动线程
        t.start();
    }
}
```

上述代码第①行是创建一个子线程，由于客户端上传的数据分为很多数据包，因此需要一个循环接收数据包，另外，调用后 receive()方法会导致线程阻塞，因此需要将接收数据的处理放到一个子线程中。

代码第②行是创建 DatagramSocket 对象，并指定端口 8080，作为服务器一般应该明确指定绑定的端口。

与 TCP Socket 不同，UDP Socket 无法知道哪些数据包已经是最后一个了，因此需要发送方发出一个特殊的数据包，包中包含了一些特殊标志。代码第③～④行是取出并判断这个标志。

客户端 UploadClient 代码如下：

```
//UploadClient.java 文件
package com.a51work6;
...
public class UploadClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");

        try ( //创建 DatagramSocket 对象,由系统分配可以使用的端口
                DatagramSocket socket = new DatagramSocket();          ①
                FileInputStream fin = new FileInputStream("./TestDir/coco2dxcplus.jpg");
                BufferedInputStream in = new BufferedInputStream(fin)) {

            //创建远程主机 IP 地址对象
            InetAddress address = InetAddress.getByName("localhost");

            //准备一个缓冲区
            byte[] buffer = new byte[1024];
            //首次从文件流中读取数据
            int len = in.read(buffer);

            while (len != -1) {
                //创建数据包对象
```

```
                DatagramPacket packet = new DatagramPacket(buffer, len, address, 8080);
                //发送数据包
                socket.send(packet);
                //再次从文件流中读取数据
                len = in.read(buffer);
            }

            //创建数据包对象
            DatagramPacket packet = new DatagramPacket("bye".getBytes(), 3, address, 8080);
            //发送结束标志
            socket.send(packet);                        ②
            System.out.println("上传完成!");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述是上传文件客户端，发送数据不会堵塞线程，因此没有使用子线程。代码第①行是创建 DatagramSocket 对象，由系统分配可以使用的端口，客户端 DatagramSocket 对象经常自己不指定。

代码第②行是发送结束标志，这个结束标志是字符串 bye，服务器端接收到这个字符串则结束接收数据包。

22.3.4 案例：聊天工具

使用 UDP Socket 将 22.2.6 节文件聊天工具重新实现一下。

服务器端 ChatServer 代码如下：

```
//ChatServer.java 文件
package com.a51work6;
…
public class ChatServer {

    public static void main(String args[]) {

        System.out.println("服务器运行…");
        //创建一个子线程
        Thread t = new Thread(() -> {                         ①
            try ( //创建 DatagramSocket 对象,指定端口 8080
                    DatagramSocket socket = new DatagramSocket(8080);
                    BufferedReader keyboardIn
                        = new BufferedReader(new InputStreamReader (System.in))) {

                while (true) {
                    /* 接收数据包 */
                    //准备一个缓冲区
```

```
                byte[] buffer = new byte[128];
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);
                //接收数据长度
                int len = packet.getLength();

                String str = new String(buffer, 0, len);
                //打印接收的数据
                System.out.printf("从客户端接收的数据:【%s】\n", str);

                /* 发送数据 */
                //从客户端传来的数据包中得到客户端地址
                InetAddress address = packet.getAddress();          ②
                //从客户端传来的数据包中得到客户端端口号
                int port = packet.getPort();                        ③

                //读取键盘输入的字符串
                String keyboardInputString = keyboardIn.readLine();
                //读取键盘输入的字节数组
                byte[] b = keyboardInputString.getBytes();
                //创建 DatagramPacket 对象,用于向客户端发送数据
                packet = new DatagramPacket(b, b.length, address, port);
                //向客户端发送数据
                socket.send(packet);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
    //启动线程
    t.start();
  }
}
```

上述代码第①行是创建一个子线程,因为 socket.receive(packet)方法会阻塞主线程。服务器给客户端发数据包,也需要知道它的 IP 地址和端口号,代码第②行是根据接收的数据包获得客户端的地址,代码第③行根据接收的数据包获得客户端的端口号。

客户端 ChatClient 代码如下:

```
//ChatClient.java 文件
package com.a51work6;
...
public class ChatClient {

    public static void main(String[] args) {

        System.out.println("客户端运行...");
        //创建一个子线程
        Thread t = new Thread(() -> {
```

```
            try ( //创建 DatagramSocket 对象,由系统分配可以使用的端口
                    DatagramSocket socket = new DatagramSocket();
                    BufferedReader keyboardIn
                         = new BufferedReader(new
InputStreamReader(System.in))) {

                while (true) {

                    /* 发送数据 */
                    //准备一个缓冲区
                    byte[] buffer = new byte[128];
                    //服务器 IP 地址
                    InetAddress address = InetAddress.getByName("localhost");
                    //服务器端口号
                    int port = 8080;
                    //读取键盘输入的字符串
                    String keyboardInputString = keyboardIn.readLine();
                    //退出循环,结束线程
                    if (keyboardInputString.equals("bye")) {
                        break;
                    }
                    //读取键盘输入的字节数组
                    byte[] b = keyboardInputString.getBytes();
                    //创建 DatagramPacket 对象
                    DatagramPacket packet = new DatagramPacket(b, b.length, address, port);
                    //发送
                    socket.send(packet);

                    /* 接收数据报 */
                    packet = new DatagramPacket(buffer, buffer.length);
                    socket.receive(packet);

                    //接收数据长度
                    int len = packet.getLength();
                    String str = new String(buffer, 0, len);
                    //打印接收的数据
                    System.out.printf("从服务器接收的数据:【%s】\n", str);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
        //启动线程
        t.start();
    }
}
```

客户端 ChatClient 代码与服务器端 ChatServer 代码类似,这里不再赘述。需要注意的是,ChatClient 可以通过键盘输入 bye 来退出循环,结束线程。

22.4 访问互联网资源

Java的java.net包中还提供了高层次网络编程类——URL，通过URL类访问互联网资源。使用URL进行网络编程，不需要对协议本身有太多的了解，相对而言是比较简单的。

22.4.1 URL概念

互联网资源是通过URL指定的，URL是Uniform Resource Locator的简称，即“一致资源定位器”，但一般都习惯简称URL。

URL组成格式如下：

```
协议名://资源名
```

“协议名”指明获取资源所使用的传输协议，如http、ftp、gopher和file等，“资源名”则应该是资源的完整地址，包括主机名、端口号、文件名或文件内部的一个引用。例如：

```
http://www.sina.com/
http://home.sohu.com/home/welcome.html
http://www.51work6.com:8800/Gamelan/network.html#BOTTOM
```

22.4.2 HTTP/HTTPS协议

访问互联网大多都基于HTTP/HTTPS协议，下面介绍一下HTTP/HTTPS协议。

1. HTTP协议

HTTP是Hypertext Transfer Protocol的缩写，即超文本传输协议。HTTP是一个属于应用层的面向对象的协议，其简捷、快速的方式适用于分布式超文本信息的传输。它于1990年提出，经过多年的使用与发展，得到不断完善和扩展。HTTP协议支持C/S网络结构，是无连接协议，即每一次请求时建立连接，服务器处理完客户端的请求后，应答给客户端，然后断开连接，不会一直占用网络资源。

HTTP/1.1协议共定义了8种请求方法：OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE和CONNECT。在HTTP访问中，一般使用GET和POST方法，其他方法都是可选的。

- GET方法：是向指定的资源发出请求，发送的信息“显式”地跟在URL后面。GET方法应该只用在读取数据，如静态图片等。GET方法有点像使用明信片给别人写信，“信内容”写在外面，接触到的人都可以看到，因此是不安全的。
- POST方法：是向指定资源提交数据，请求服务器进行处理，例如提交表单或者上传文件等。数据被包含在请求体中。POST方法像是把“信内容”装入信封中，接触到的人都看不到，因此是安全的。

2. HTTPS协议

HTTPS是HyperText Transfer Protocol Secure的缩写，即超文本传输安全协议，是超

文本传输协议和 SSL 的组合，用以提供加密通信及对网络服务器身份的鉴定。

简单地说，HTTPS 是 HTTP 的升级版，HTTPS 与 HTTP 的区别是：HTTPS 使用 https://代替 http://，HTTPS 使用端口 443，而 HTTP 使用端口 80 来与 TCP/IP 进行通信。SSL 使用 40 位关键字作为 RC4 流加密算法，这对于商业信息的加密是合适的。HTTPS 和 SSL 支持使用 X.509 数字认证，如果需要，用户可以确认发送者是谁。

22.4.3 使用 URL 类

Java 的 java.net.URL 类用于请求互联网上的资源，采用 HTTP/HTTPS 协议，请求方法是 GET 方法，一般是请求静态的、少量的服务器端数据。

URL 类常用构造方法如下。

- URL(String spec)：根据字符串表示形式创建 URL 对象。
- URL(String protocol, String host, String file)：根据指定的协议名、主机名和文件名称创建 URL 对象。
- URL(String protocol, String host, int port, String file)：根据指定的协议名、主机名、端口号和文件名称创建 URL 对象。

URL 类常用方法如下。

- InputStream openStream()：打开到此 URL 的连接，并返回一个输入流。
- URLConnection openConnection()：打开到此 URL 的新连接，返回一个 URLConnection 对象。

下面通过一个示例介绍如何使用 java.net.URL 类。示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;
…
public class HelloWorld {

    public static void main(String[] args) {
        //Web 网址
        String url = "http://www.sina.com.cn/";

        URL reqURL;
        try {
            reqURL = new URL(url);                      ①
        } catch (MalformedURLException e1) {
            return;
        }

        try ( //打开网络通信输入流
                InputStream is = reqURL.openStream();     ②
                InputStreamReader isr = new InputStreamReader(is, "utf-8");
                BufferedReader br = new BufferedReader(isr)) {

            StringBuilder sb = new StringBuilder();
            String line = br.readLine();
            while (line != null) {
```

```
                sb.append(line);
                sb.append('\n');
                line = br.readLine();
            }
            //日志输出
            System.out.println(sb);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码第①行创建 URL 对象,参数是一个 HTTP 网址。代码第②行通过 URL 对象的 openStream()方法打开输入流。

22.4.4 案例:Downloader

为了进一步熟悉 URL 类,本节介绍一个下载程序 Downloader。Downloader.java 代码如下:

```
//Downloader.java 文件
package com.a51work6;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class Downloader {

    //Web 服务网址
     private static String urlString = "https://ss0.bdstatic.com/5aV1bjqh_Q23odCf/" +
"static/superman/img/logo/bd_logo1_31bdc765.png";

    public static void main(String[] args) {
        download();
    }

    //下载方法
    private static void download() {

        HttpURLConnection conn = null;

        try {
```

```
            //创建 URL 对象
            URL reqURL = new URL(urlString);
            //打开连接
            conn = (HttpURLConnection) reqURL.openConnection();                ①

            try (//从连接对象获得输入流
                    InputStream is = conn.getInputStream();                    ②
                    BufferedInputStream bin = new BufferedInputStream (is);     ③
                    //创建文件输出流
                    OutputStream os = new FileOutputStream("./download.png");  ④
                    BufferedOutputStream bout = new
BufferedOutputStream (os);) {                                                  ⑤

                byte[] buffer = new byte[1024];
                int bytesRead = bin.read(buffer);
                while (bytesRead != -1) {
                    bout.write(buffer, 0, bytesRead);
                    bytesRead = bin.read(buffer);
                }
            } catch (IOException e) {
            }
            System.out.println("下载完成.");
        } catch (IOException e) {
        } finally {
            if (conn != null) {
                conn.disconnect();
            }
        }
    }
}
```

上述代码第①行打开连接获得 HttpURLConnection 对象，代码第②行是从连接对象获得输入流，代码第③行创建缓冲流输入流，使用缓冲流可以提高读写效率。

代码第④行是创建文件输出流，代码第⑤行是创建缓冲流输出流。

运行 Downloader 程序，如果成功，则会在当前目录获得一张图片。

本章小结

本章主要介绍了 Java 网络编程。首先介绍了一些网络方面的基本知识。然后重点介绍了 TCP Socket 编程和 UDP Socket 编程，其中 TCP Socket 编程很有代表性，希望重点掌握这部分知识。最后介绍了使用 URL 类访问互联网资源。

22.5 同步练习

1. 下列选项中哪些类可以用来实现 TCP/IP 客户服务器程序？(　　)

A. ServerSocket　　B. Server　　C. Socket

D. DatagramPacket　　E. DatagramSocket

2. 下列选项中正确地创建 Socket 的语句有(　　)。

A. Socket a = new Socket(80);

B. Socket b = new Socket("130.3.4.5",80);

C. ServerSocket c = new Socket(80);

D. ServerSocket d = new Socket("130.3.4.5",80);

3. 下列选项中正确的论述是(　　)。

A. ServerSocket.accept 是阻塞的

B. BufferedReader.readLine 是阻塞的

C. DatagramSocket.receive 是阻塞的

D. DatagramSocket.send 是阻塞的

4. 下面的语句创建一 DatagramSocket 对象,(　　)是正确的。

A. DatagramSocket a = new DatagramSocket();

B. DatagramSocket b = new DatagramSocket(80);

C. DatagramSocket c = new DatagramSocket("127.0.0.1",70);

D. DatagramSocket d = new DatagramSocket("127.0.0.1");

第 23 章 CHAPTER 23 Swing 图形用户界面编程

图形用户界面(Graphical User Interface,GUI)编程对于某种语言来说非常重要。Java 的应用主要方向是基于 Web 浏览器的应用,用户界面主要是 HTML、CSS 和 JavaScript 等基于 Web 的技术,这些技术要到 Java EE 阶段才能学习到。

而本章介绍的 Java 图形用户界面技术是基于 Java SE 的 Swing,事实上 Swing 在实际应用中使用并不广泛,因此本章的内容可只做了解。

23.1 Java 图形用户界面技术

Java 图形用户界面技术主要有 AWT、Applet、Swing 和 Java FX。

1. AWT

AWT(Abstract Window Toolkit)是抽象窗口工具包,AWT 是 Java 程序提供的建立图形用户界面最基础的工具集。AWT 支持图形用户界面编程的功能包括用户界面组件(控件)、事件处理模型、图形图像处理(形状和颜色)、字体、布局管理器和本地平台的剪贴板来进行剪切和粘贴等。AWT 是 Applet 和 Swing 技术的基础。

AWT 在实际的运行过程中是调用所在平台的图形系统,因此同样一段 AWT 程序在不同的操作系统平台下运行时所看到的样式是不同的。例如在 Windows 下运行,则显示的窗口是 Windows 风格的窗口,如图 23-1 所示,而在 UNIX 下运行时,则显示的是 UNIX 风格的窗口,如图 23-2 所示为 macOS① 风格的 AWT 窗口。

2. Applet

Applet 称为 Java 小应用程序,Applet 基础是 AWT,但它主要嵌入到 HTML 代码中,由浏览器加载和运行。由于存在安全隐患和运行速度慢等问题,已经很少使用。

3. Swing

Swing 是 Java 主要的图形用户界面技术,Swing 提供跨平台的界面风格,用户可以自定义 Swing 的界面风格。Swing 提供了比 AWT 更完整的组件,引入了许多新的特性。Swing API 是围绕着实现 AWT 各个部分的 API 构筑的。Swing 是由 100%纯 Java 实现的,Swing 组件没有本地代码,不依赖操作系统的支持,这是它与 AWT 组件的最大区别。本章重点介绍 Swing 技术。

---

① macOS 是苹果计算机操作系统,它也是 UNIX 内核。

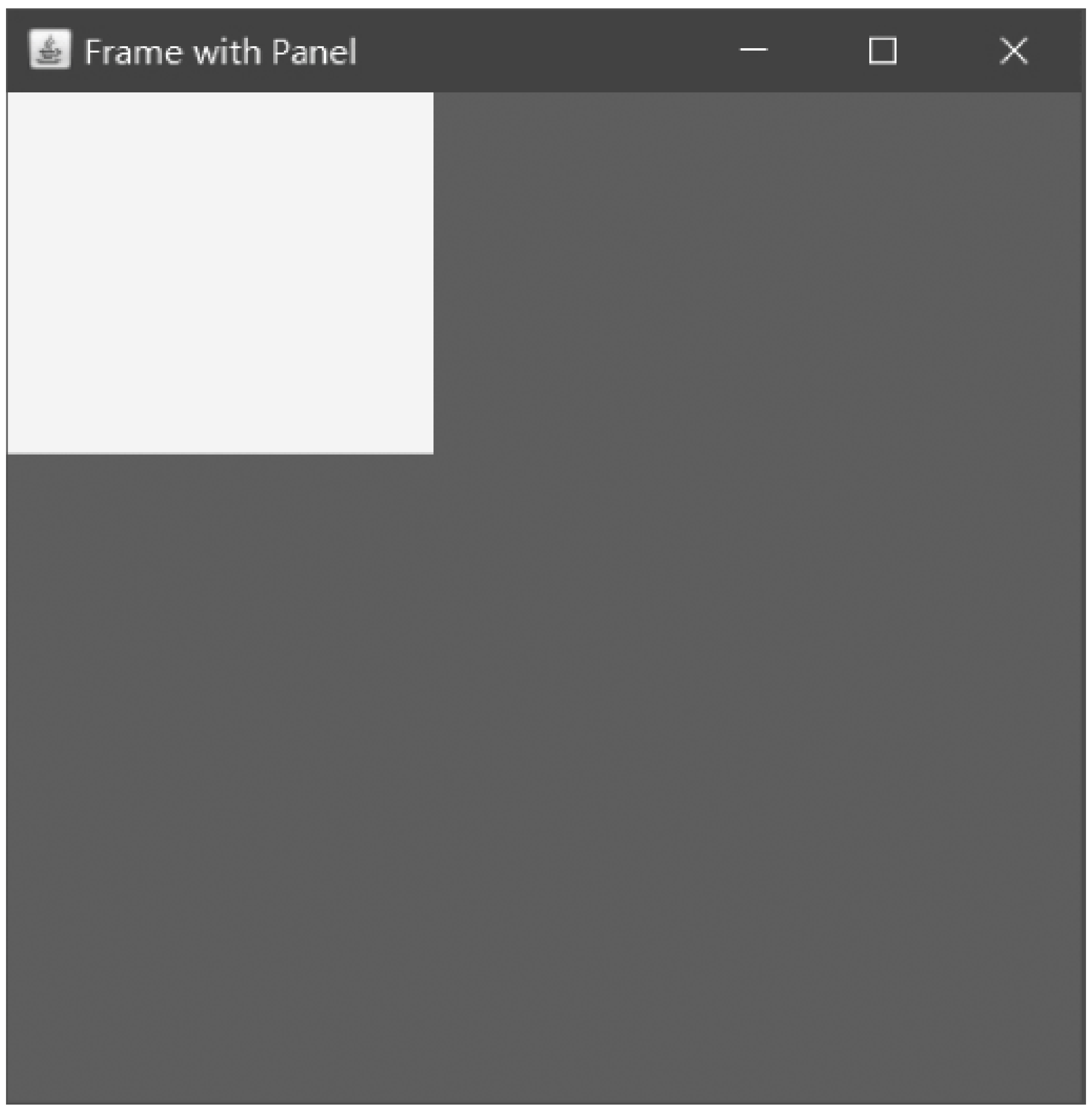


图 23-1 Windows 风格的 AWT 窗口



图 23-2 macOS 风格的 AWT 窗口

4. Java FX

Java FX 是开发丰富互联网应用程序(Rich Internet Application,RIA)的图形用户界面技术,Java FX 期望能够在桌面应用的开发领域与 Adobe 公司的 AIR、微软公司的 Silverlight 相竞争。传统的互联网应用程序基于 Web 的,客户端是浏览器。而丰富互联网应用程序试图打造自己的客户端,替代浏览器。

23.2 Swing 技术基础

AWT 是 Swing 的基础，Swing 事件处理和布局管理都是依赖于 AWT，AWT 内容来自 java.awt 包，Swing 内容来自 javax.swing 包。AWT 和 Swing 作为图形用户界面技术包括 4 个主要的概念：组件（Component）、容器（Container）、事件处理和布局管理器（LayoutManager）。下面将围绕这些概念展开。

23.2.1 Swing 类层次结构

容器和组件构成了 Swing 的主要内容，下面分别介绍一下 Swing 中容器和组件类层次结构。

图 23-3 所示是 Swing 容器类层次结构。Swing 容器类主要有 JWindow、JFrame 和 JDialog，其他的不带“J”开头都是 AWT 提供的类，在 Swing 中大部分类都是以“J”开头。

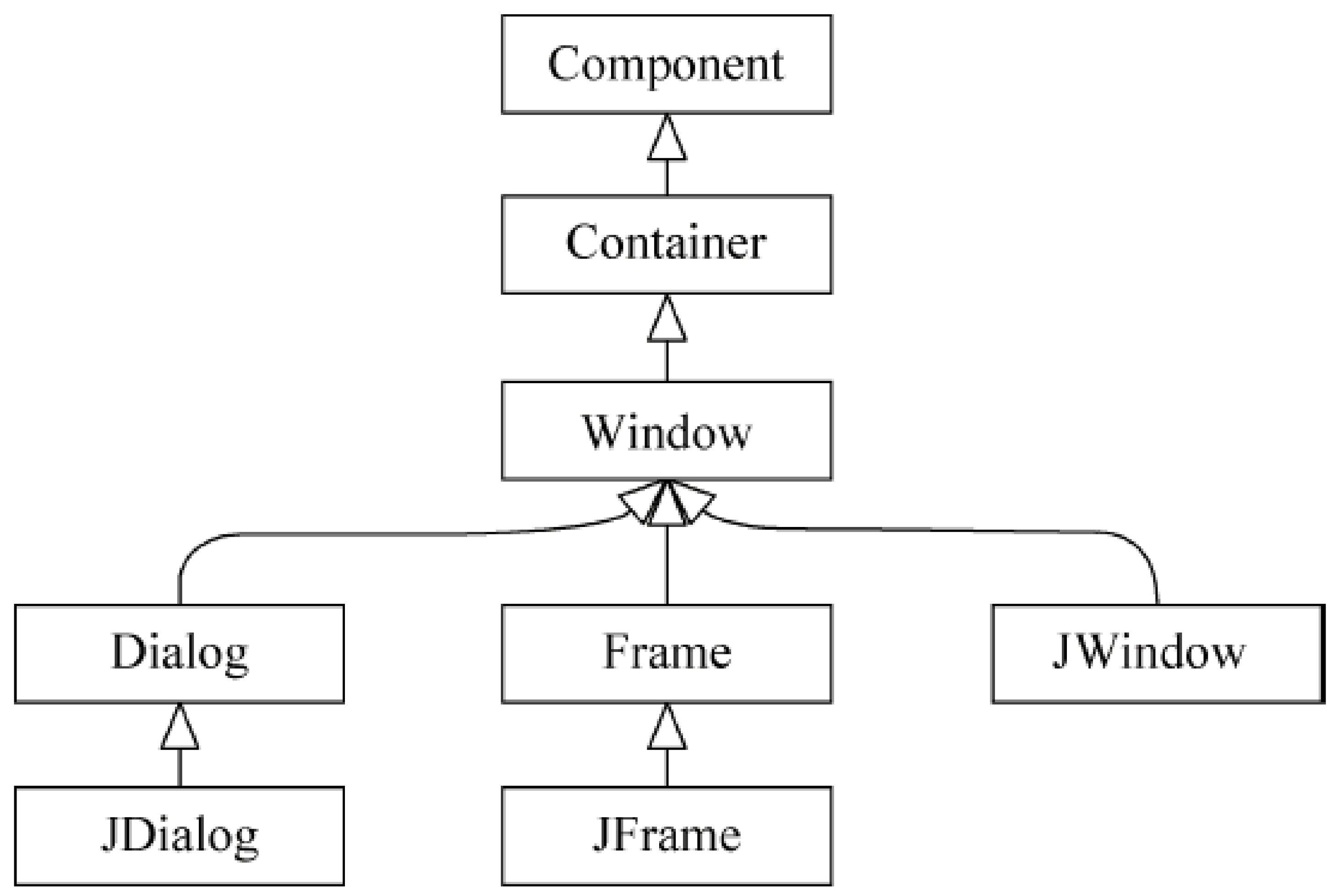


图 23-3 Swing 容器类层次结构

图 23-4 所示是 Swing 组件类层次结构。Swing 所有组件继承自 JComponent，JComponent 又间接继承自 AWT 的 java.awt.Component 类。Swing 组件很多，这里不一一解释，在后面的学习过程中会重点介绍组件。

23.2.2 Swing 程序结构

图形用户界面主要是由窗口以及窗口中的组件构成的，编写 Swing 程序主要就是创建窗口和添加组件的过程。Swing 中的窗口主要是使用 JFrame，很少使用 JWindow。JFrame 有标题、边框、菜单、大小和窗口管理按钮等窗口要素，而 JWindow 没有标题栏和窗口管理按钮。

构建 Swing 程序主要有两种方式：创建 JFrame 或继承 JFrame。下面通过一个示例介绍一下这两种方式如何实现。该示例运行效果如图 23-5 所示，窗口标题是 MyFrame，窗口中有显示字符串"Hello Swing!"。

1. 创建 JFrame 方式

创建 JFrame 方式就是直接实例化 JFrame 对象，然后设置 JFrame 属性，添加窗口所需要的组件。

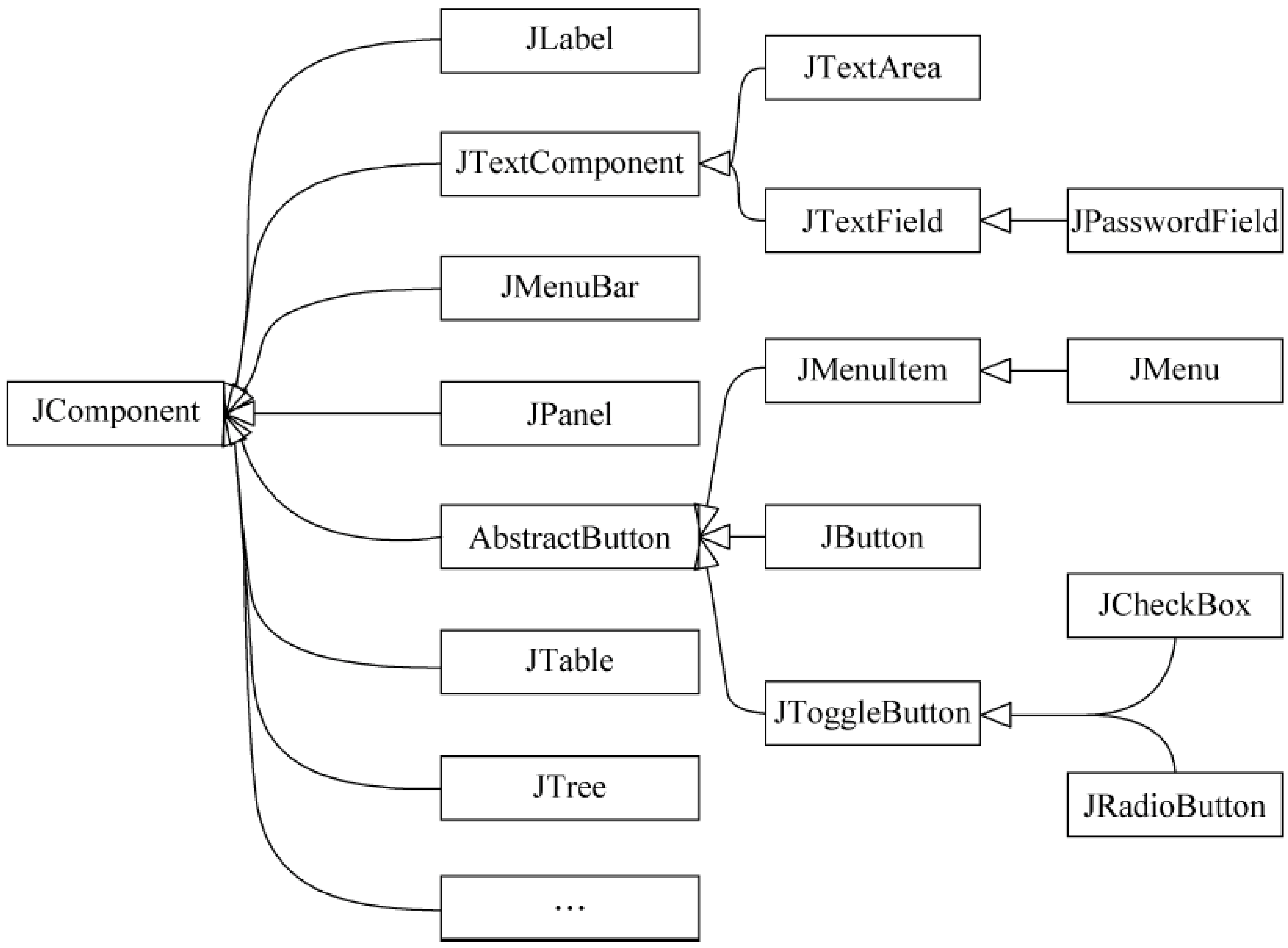


图 23-4　Swing 组件类层次结构

示例代码如下：

```
//SwingDemo1.java 文件
package com.a51work6;

import java.awt.Container;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class SwingDemo1{

    public static void main(String[] args) {
        //创建窗口对象
        JFrame frame = new JFrame("MyFrame");                ①

        //创建标签
        JLabel label = new JLabel("Hello Swing!");           ②
        //获得窗口的内容面板
        Container contentPane = frame.getContentPane();      ③
        //添加标签到内容面板
        contentPane.add(label);                              ④

        //设置窗口大小
        frame.setSize(300, 300);                             ⑤
        //设置窗口可见
        frame.setVisible(true);                              ⑥
    }
}
```



图 23-5　Swing 示例运行效果

上述代码第①行使用JFrame的JFrame(String title)构造方法创建JFrame对象，title是设置创建的标题。默认情况下，JFrame是没有大小且不可见的，因此创建JFrame对象后还需要设置大小和可见。代码第⑤行是设置窗口的大小，代码第⑥行是设置窗口的可见。

注意 设置JFrame窗口的大小和可见这两条语句应该在添加完成所有组件之后调用。否则在多个组件情况下，会导致有些组件没有显示。

创建好窗口后，就需要将其中的组件添加进来。代码第②行是创建标签对象，构造方法中字符串参数是标签要显示的文本。创建好组件之后需要把它添加到窗口的内容面板上。代码第③行是获得窗口的内容面板，它是Container容器类型。代码第④行是调用容器的add()方法将组件添加到窗口上。

注意 在Swing中添加到JFrame上的所有可见组件，除菜单栏外，全部添加到内容面板上，而不要直接添加到JFrame上，这是Swing绘制系统所要求的。内容面板是JFrame中包含的一个子容器，如图23-6所示。

提示 几乎所有的图形用户界面技术在构建界面时都采用层级结构(树状结构)，如图23-7所示。根是顶级容器(只能包含其他容器的容器)，子容器有内容面板和菜单栏(本例中没有菜单)，然后其他的组件添加到内容面板容器中。所有的组件都有add()方法，通过调用add()方法将其他组件添加到容器中，作为当前容器的子组件。

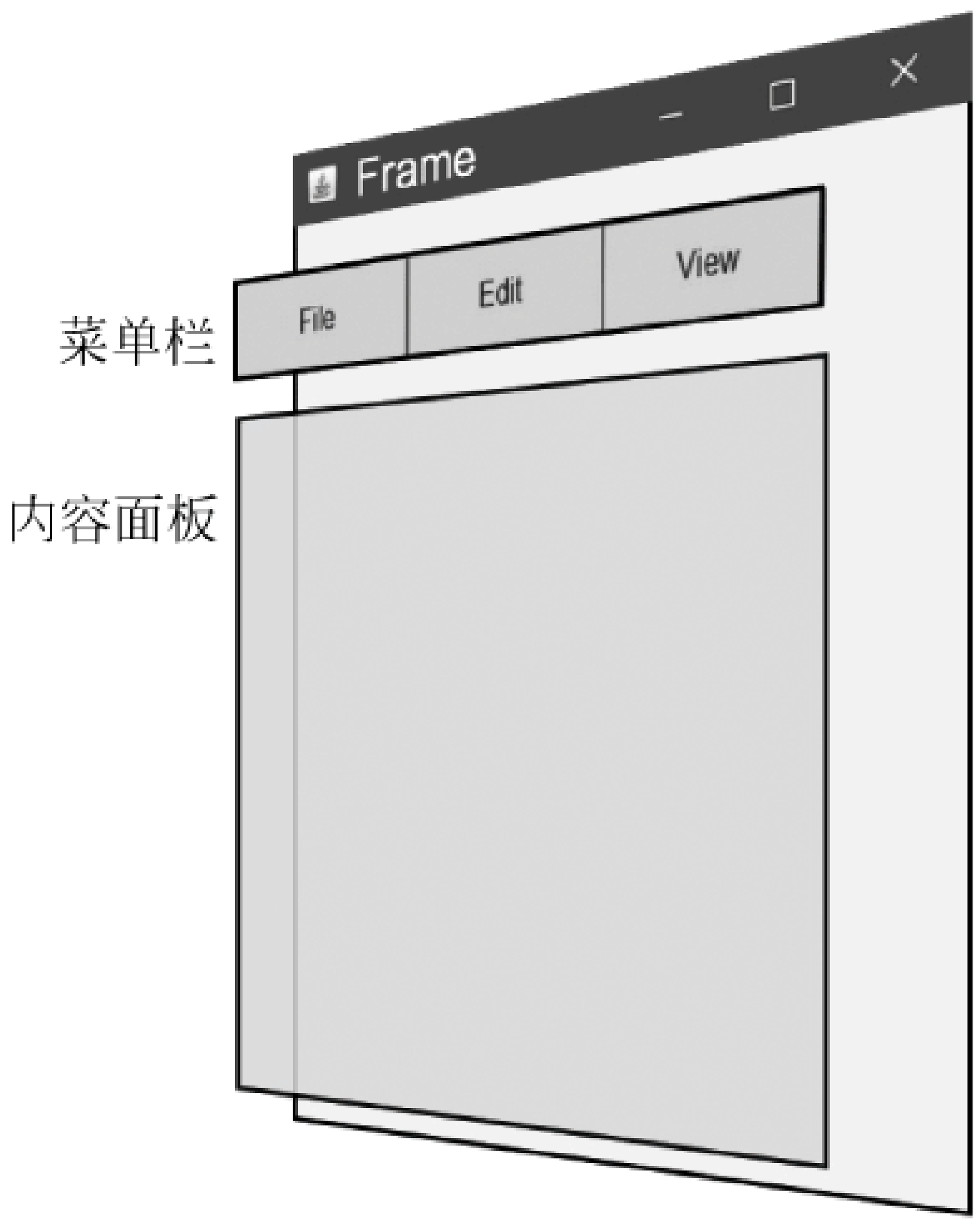


图23-6 JFrame的内容面板

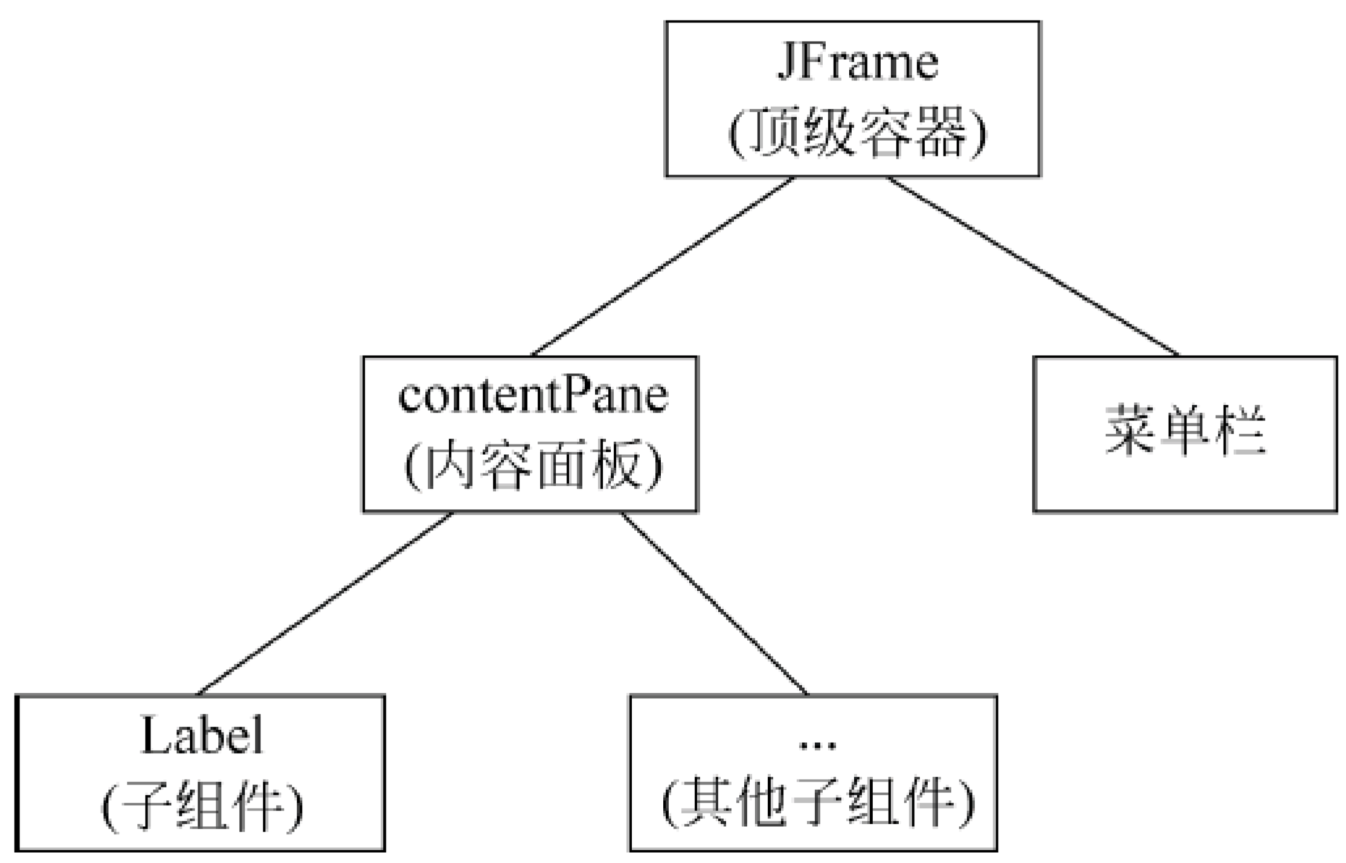


图23-7 界面构建层次

2. 继承JFrame方式

继承JFrame方式就是编写一个继承JFrame的子类，在构造方法中初始化窗口，添加窗口所需要的组件。

自定义窗口代码如下：

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.Container;

import javax.swing.JFrame;
```

```
import javax.swing.JLabel;

public class MyFrame extends JFrame {                    ①

    public MyFrame(String title) {                       ②
        super(title);

        //创建标签
        JLabel label = new JLabel("Hello Swing!");
        //获得窗口的内容面板
        Container contentPane = getContentPane();
        //添加标签到内容面板
        contentPane.add(label);

        //设置窗口大小
        setSize(300, 300);
        //设置窗口可见
        setVisible(true);
    }
}
```

上述代码第①行是声明 MyFrame 继承 JFrame，代码第②行定义构造方法，参数是窗口标题。

调用代码如下：

```
//SwingDemo2.java 文件
package com.a51work6;

public class SwingDemo2{

    public static void main(String[] args) {
        //创建窗口对象
        new MyFrame("MyFrame");
    }
}
```

运行上述代码可以发现，继承 JFrame 方式和创建 JFrame 方式效果完全一样。

提示 创建 JFrame 方式适合于小项目，代码量少、窗口不多、组件少的情况。继承 JFrame 方式，适合于大项目，可以针对不同界面自定义一个 Frame 类，属性可以在构造方法中进行设置；缺点是有很多类文件需要有效地管理。

23.3 事件处理模型

图形界面的组件要响应用户操作，就必须添加事件处理机制。Swing 采用 AWT 的事件处理模型进行事件处理。在事件处理的过程中涉及三个要素。

（1）事件。是用户对界面的操作，在 Java 中事件被封装称为事件类 java.awt.AWTEvent 及其子类，例如按钮单击事件类是 java.awt.event.ActionEvent。

(2) 事件源。是事件发生的场所，就是各个组件，例如按钮单击事件的事件源是按钮(Button)。

(3) 事件处理者。是事件处理程序，在 Java 中事件处理者是实现特定接口的事件对象。

在事件处理模型中最重要的是事件处理者，它根据事件(假设 XXXEvent 事件)的不同会实现不同的接口，这些接口命名为 XXXListener，所以事件处理者也称为事件监听器。最后事件源通过 addXXXListener()方法添加事件监听，监听 XXXEvent 事件。各种事件和对应的监听器接口如表 23-1 所示。

表 23-1 事件类型和事件监听器接口

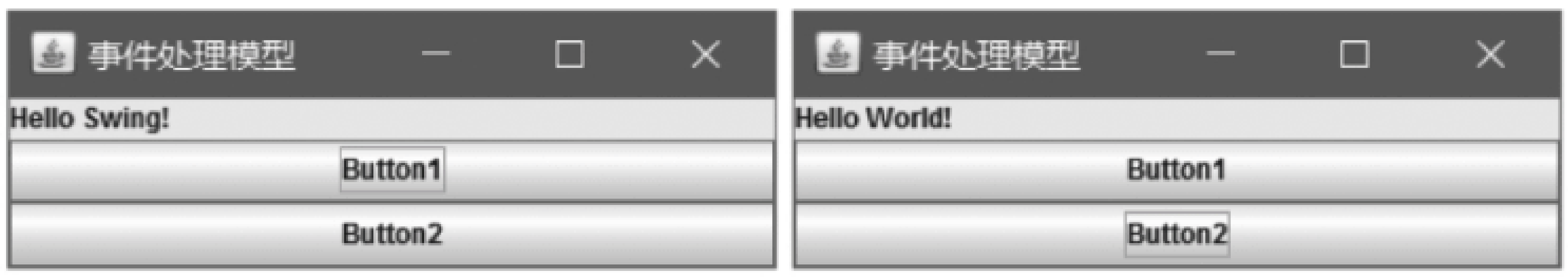
事件类型	相应监听器接口	监听器接口中的方法
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse	MouseListener	mousePressed(MouseEvent)
		mouseReleased(MouseEvent)
		mouseEntered(MouseEvent)
		mouseExited(MouseEvent)
		mouseClicked(MouseEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent)
		mouseMoved(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent)
		keyReleased(KeyEvent)
		keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent)
		focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent)
		componentHidden(ComponentEvent)
		componentResized(ComponentEvent)
		componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent)
		windowOpened(WindowEvent)
		windowIconified(WindowEvent)
		windowDeiconified(WindowEvent)
		windowClosed(WindowEvent)
		windowActivated(WindowEvent)
		windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

事件处理者可以实现 XXXListener 接口任何形式，即外部类、内部类、匿名内部类和 Lambda 表达式。如果 XXXListener 接口只有一个抽象方法，事件处理者还可以是 Lambda 表达式。为了访问窗口中的组件方便，往往使用内部类、匿名内部类和 Lambda 表达式情况很多。

23.3.1　采用内部类处理事件

内部类和匿名内部类能够方便访问窗口中的组件，所以这里重点介绍内部类和匿名内部类实现的事件监听器。

下面通过一个示例介绍采用内部类和匿名内部类实现的事件处理模型。如图23-8所示的示例，界面中有两个按钮和一个标签，当单击Button1或Button2时会改变标签显示的内容。



(a)　　　　(b)

图23-8　事件处理模型示例

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {

    //声明标签
    JLabel label;                                                  ①

    public MyFrame(String title) {
        super(title);

        //创建标签
        label = new JLabel("Label");
        //添加标签到内容面板
        getContentPane().add(label, BorderLayout.NORTH);           ②

        //创建 Button1
        JButton button1 = new JButton("Button1");
        //添加 Button1 到内容面板
        getContentPane().add(button1, BorderLayout.CENTER);        ③

        //创建 Button2
        JButton button2 = new JButton("Button2");
```

```
        //添加 Button2 到内容面板
        getContentPane().add(button2, BorderLayout.SOUTH);            ④

        //设置窗口大小
        setSize(350, 120);
        //设置窗口可见
        setVisible(true);

        //注册事件监听器,监听 Button2 单击事件
        button2.addActionListener(new ActionEventHandler());          ⑤

        //注册事件监听器,监听 Button1 单击事件
        button1.addActionListener(new ActionListener() {              ⑥
            @Override
            public void actionPerformed(ActionEvent event) {
                label.setText("Hello Swing!");
            }
        });
    }

    //Button2 事件处理者
    class ActionEventHandler implements ActionListener {              ⑦
        @Override
        public void actionPerformed(ActionEvent e) {
            label.setText("Hello World!");
        }
    }
}
```

上述代码第②行通过 add(label, BorderLayout.NORTH)方法将标签添加到内容面板,这个 add()方法与前面介绍的有所不同,它的第二个参数是指定组件的位置。有关布局管理的内容将在 23.4 节详细介绍。类似的添加还有第③行和第④行。

代码第⑤行和第⑥行都是注册事件监听器监听 Button 的单击事件。但是第⑤行的事件监听器是一个内部类 ActionEventHandler,它的定义是在代码第⑦行。代码第⑥行的事件监听器是一个匿名内部类。

提示　在事件处理模型中,对于内部类实现的模型,内部类会定义为成员内部类,因此不能访问其他方法中的局部变量组件,只能访问成员变量组件,所以代码第①行将标签组件声明为成员变量,否则 ActionEventHandler 内部类无法访问该组件。而匿名内部类既可以访问所在方法的局部变量组件,也可以访问成员变量组件。

23.3.2　采用 Lambda 表达式处理事件

如果一个事件监听器接口只有一个抽象方法,则可以使用 Lambda 表达式实现事件处理,这些接口主要有 ActionListener、AdjustmentListener、ItemListener、MouseWheelListener、TextListener 和 WindowStateListener 等。

下面将 23.3.1 节的示例修改如下:

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame implements ActionListener {                ①

    //声明标签
    JLabel label;

    public MyFrame(String title) {
        super(title);

        //创建标签
        label = new JLabel("Label");
        //添加标签到内容面板
        getContentPane().add(label, BorderLayout.NORTH);

        //创建 Button1
        JButton button1 = new JButton("Button1");
        //添加 Button1 到内容面板
        getContentPane().add(button1, BorderLayout.CENTER);

        //创建 Button2
        JButton button2 = new JButton("Button2");
        //添加 Button2 到内容面板
        getContentPane().add(button2, BorderLayout.SOUTH);

        //设置窗口大小
        setSize(350, 120);
        //设置窗口可见
        setVisible(true);

        //注册事件监听器,监听 Button2 单击事件
        button2.addActionListener(this);                                    ②

        //注册事件监听器,监听 Button1 单击事件
        button1.addActionListener((event) -> {                              ③
            label.setText("Hello World!");
        });
    }

    @Override
    public void actionPerformed(ActionEvent event) {                        ④
        label.setText("Hello Swing!");
    }
}
```

上述代码第③行采用 Lambda 表达式实现事件监听器,可见代码非常简单。另外,当前窗口本身也可以是事件处理者,代码第①行声明窗口实现 ActionListener 接口。代码第④行是实现抽象方法,那么注册事件监听器参数就是 this,见代码第②行。

23.3.3 使用适配器

事件监听器都是接口,在 Java 接口中定义的抽象方法必须全部实现,哪怕对某些方法并不关心,也要给一对空的大括号表示实现。例如,WindowListener 是窗口事件(WindowEvent)监听器接口,为了在窗口中接收到窗口事件,需要在窗口中注册 WindowListener 事件监听器。示例代码如下:

```
this.addWindowListener(new WindowListener() {

    @Override
    public void windowActivated(WindowEvent e) {
    }

    @Override
    public void windowClosed(WindowEvent e) {
    }

    @Override
    public void windowClosing(WindowEvent e) {                    ①
        //退出系统
        System.exit(0);
    }

    @Override
    public void windowDeactivated(WindowEvent e) {
    }

    @Override
    public void windowDeiconified(WindowEvent e) {
    }

    @Override
    public void windowIconified(WindowEvent e) {
    }

    @Override
    public void windowOpened(WindowEvent e) {
    }
});
```

实现 WindowListener 接口需要提供它 7 个方法的实现,很多情况下只是想在关闭窗口时释放一下资源,只需要实现代码第①行的 windowClosing(WindowEvent e),其他的方法并不关心,但是也必须给出空的实现。这样的代码看起来很臃肿,为此 Java 还提供了一些与监听器相配套的适配器。监听器是接口,命名采用 XXXListener,而适配器是类,命名采

用 XXX Adapter。在使用时通过继承事件所对应的适配器类覆盖所需要的方法，无关方法不用实现。

采用适配器注册接收窗口事件代码如下：

```
this.addWindowListener(new WindowAdapter(){
    @Override
    public void windowClosing(WindowEvent e) {
        //退出系统
        System.exit(0);
    }
});
```

可见代码非常简洁。事件适配器提供了一种简单的实现监听器的手段，可以缩短程序代码。但是，由于 Java 的单一继承机制，当需要多种监听器或此类已有父类时，就无法采用事件适配器。

并非所有的监听器接口都有对应的适配器类，一般定义了多个方法的监听器接口。例如，WindowListener 有多个方法对应多种不同的窗口事件时，才需要配套的适配器。主要的适配器如下。

- ComponentAdapter：组件适配器。
- ContainerAdapter：容器适配器。
- FocusAdapter：焦点适配器。
- KeyAdapter：键盘适配器。
- MouseAdapter：鼠标适配器。
- MouseMotionAdapter：鼠标运动适配器。
- WindowAdapter：窗口适配器。

23.4 布局管理

为了实现图形用户界面的跨平台，并实现动态布局等效果，Java 将容器内的所有组件布局交给布局管理器管理。布局管理器负责组件的排列顺序、大小、位置，以及当窗口移动或调整大小后组件如何变化等。

Java SE 提供了 7 种布局管理器，包括 FlowLayout、BorderLayout、GridLayout、BoxLayout、CardLayout、SpringLayout 和 GridBagLayout，其中最基础的是 FlowLayout、BorderLayout 和 GridLayout 布局管理器。下面重点介绍这 3 种布局。

23.4.1 FlowLayout 布局

FlowLayout 布局摆放组件的规律是：从上到下、从左到右进行摆放，如果容器足够宽，第一个组件先添加到容器中第一行的最左边，后续的组件依次添加到上一个组件的右边，如果当前行已摆放不下该组件，则摆放到下一行的最左边。

FlowLayout 主要的构造方法如下。

- FlowLayout(int align, int hgap, int vgap)：创建一个 FlowLayout 对象，它具有指

定的对齐方式以及指定的水平和垂直间隙，hgap参数是组件之间的水平间隙，vgap参数是组件之间的垂直间隙，单位是像素。

- FlowLayout(int align)：创建一个FlowLayout对象，指定的对齐方式，默认的水平和垂直间隙是5个单位。
- FlowLayout：创建一个FlowLayout对象，它是居中对齐的，默认的水平和垂直间隙是5个单位。

上述参数align是对齐方式，它是通过FlowLayout的常量指定的。这些常量说明如下。

- FlowLayout.CENTER：指示每一行组件都应该是居中的。
- FlowLayout.LEADING：指示每一行组件都应该与容器方向的开始边对齐，例如，对于从左到右的方向，则与左边对齐。
- FlowLayout.LEFT：指示每一行组件都应该是左对齐的。
- FlowLayout.RIGHT：指示每一行组件都应该是右对齐的。
- FlowLayout.TRAILING：指示每行组件都应该与容器方向的结束边对齐，例如，对于从左到右的方向，则与右边对齐。

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MyFrame extends JFrame {

    //声明标签
    JLabel label;

    public MyFrame(String title) {
        super(title);

        setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));      ①
        //创建标签
        label = new JLabel("Label");
        //添加标签到内容面板
        getContentPane().add(label);                             ②

        //创建 Button1
        JButton button1 = new JButton("Button1");
        //添加 Button1 到内容面板
        getContentPane().add(button1);                           ③

        //创建 Button2
```

```
        JButton button2 = new JButton("Button2");
        //添加 Button2 到内容面板
        getContentPane().add(button2);                ④

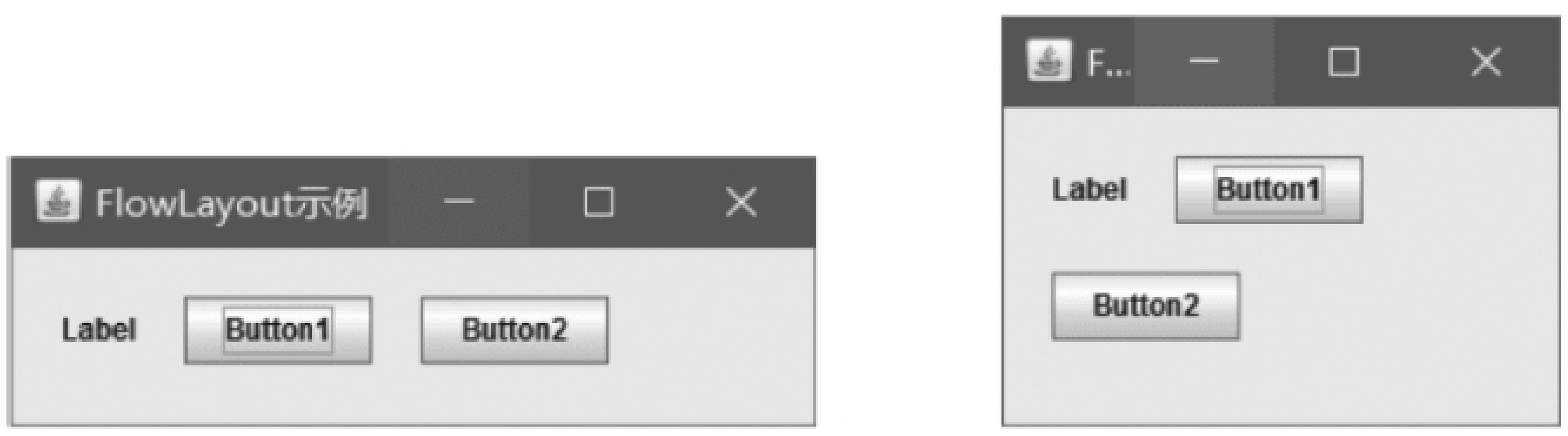
        //设置窗口大小
        setSize(350, 120);
        //设置窗口可见
        setVisible(true);

        //注册事件监听器,监听 Button2 单击事件
        button2.addActionListener((event) -> {
            label.setText("Hello Swing!");
        });

        //注册事件监听器,监听 Button1 单击事件
        button1.addActionListener((event) -> {
            label.setText("Hello World!");
        });
    }
}
```

上述代码第①行设置当前窗口的布局是 FlowLayout 布局,采用 FlowLayout(int align, int hgap, int vgap)构造方法。一旦设置了 FlowLayout 布局,就可以通过 add(Component comp)方法添加组件到窗口的内容面板,见代码第②~④行。

运行结果如图 23-9(a)所示。采用 FlowLayout 布局如果水平空间比较小,组件会垂直摆放,拖曳窗口的边缘使窗口变窄,如图 23-9(b)所示,最后一个组件换行。



(a) (b)

图 23-9 FlowLayout 示例运行结果

23.4.2 BorderLayout 布局

BorderLayout 布局是窗口的默认布局管理器,23.3 节的示例就是采用 BorderLayout 布局实现的。

BorderLayout 是 JWindow、JFrame 和 JDialog 的默认布局管理器。BorderLayout 布局管理器把容器分成 5 个区域:北、南、东、西、中,如图 23-10 所示,每个区域只能放置一个组件。

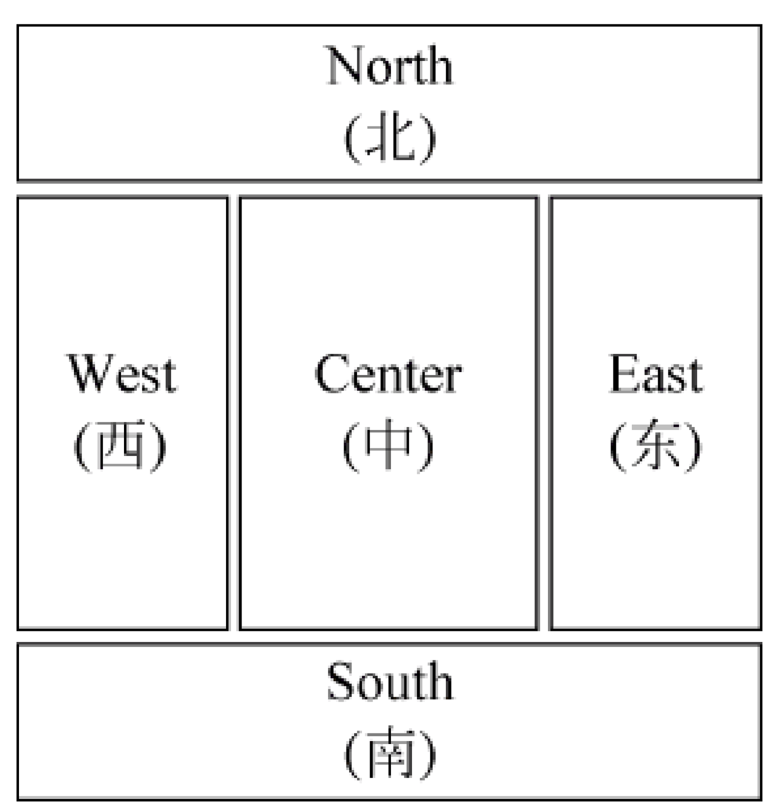


图 23-10 BorderLayout 布局

BorderLayout 主要的构造方法如下。

- BorderLayout(int hgap, int vgap):创建一个 BorderLayout 对象,指定水平和垂直间隙,hgap 参数是组件之间的水平

间隙，vgap 参数是组件之间的垂直间隙，单位是像素。

- BorderLayout()：创建一个 BorderLayout 对象，组件之间没有间隙。

BorderLayout 布局有 5 个区域，为此 BorderLayout 中定义了 5 个约束常量，说明如下。

- BorderLayout. CENTER：中间区域的布局约束(容器中央)。
- BorderLayout. EAST：东区域的布局约束(容器右边)。
- BorderLayout. NORTH：北区域的布局约束(容器顶部)。
- BorderLayout. SOUTH：南区域的布局约束(容器底部)。
- BorderLayout. WEST：西区域的布局约束(容器左边)。

示例代码如下。

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.BorderLayout;
import java.awt.Button;

import javax.swing.JFrame;

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);

        //设置 BorderLayout 布局
        setLayout(new BorderLayout(10, 10));                              ①

        //添加按钮到容器的 North 区域
        getContentPane().add(new Button("北"), BorderLayout.NORTH);       ②
        //添加按钮到容器的 South 区域
        getContentPane().add(new Button("南"), BorderLayout.SOUTH);       ③
        //添加按钮到容器的 East 区域
        getContentPane().add(new Button("东"), BorderLayout.EAST);        ④
        //添加按钮到容器的 West 区域
        getContentPane().add(new Button("西"), BorderLayout.WEST);        ⑤
        //添加按钮到容器的 Center 区域
        getContentPane().add(new Button("中"), BorderLayout.CENTER);      ⑥

        setSize(300, 300);
        setVisible(true);
    }
}
```

上述代码第①行设置窗口布局为 BorderLayout 布局，组件之间的间隙是 10 个像素，事实上窗口默认布局就是 BorderLayout，只是组件之间没有间隙，如图 23-11 所示。代码第②～⑥行分别添加了 5 个按钮，使用的添加方法是 add(Component comp, Object constraints)，第二个参数 constraints 是指定约束。

当使用 BorderLayout 时，如果容器的大小发生变化，其变化规律为：组件的相对位置不变，大小发生变化。如图 23-12 所示，如果容器变高或矮，则北和南不变，西、中和东变高或矮；如果容器变宽或窄，西和东区域不变，北、中和南变宽或窄。

另外，在 5 个区域中不一定都放置了组件，如果某个区域缺少组件，界面布局会有比较大的影响，具体影响如图 23-13 所示，其中列出了主要的一些情况。

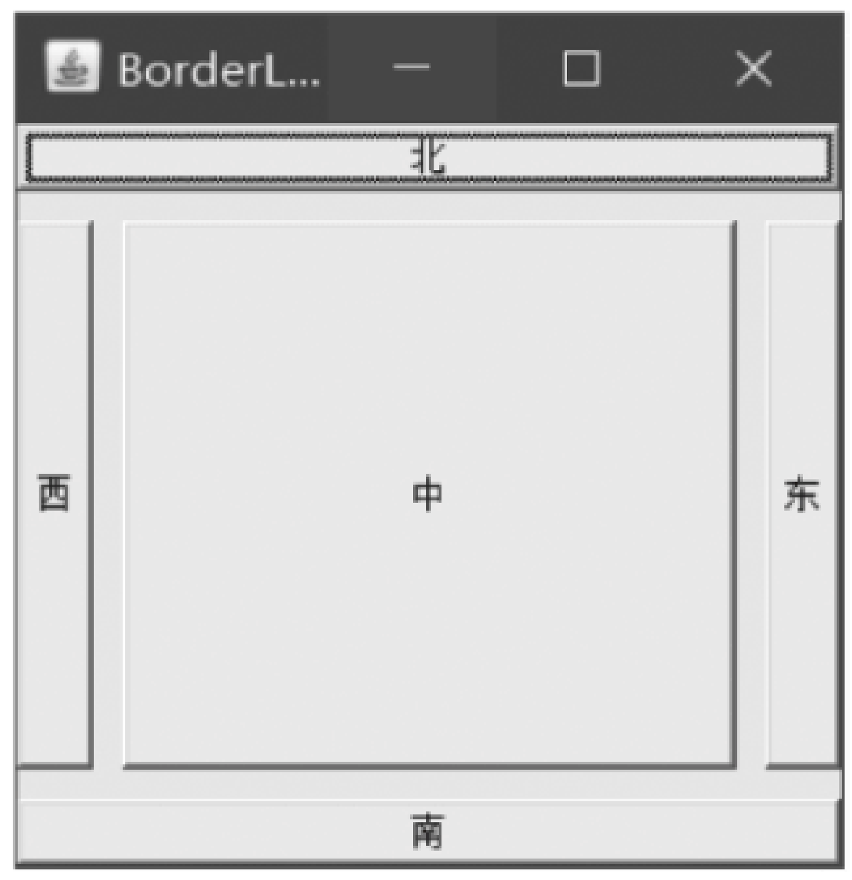


图 23-11 BorderLayout 布局示例运行结果

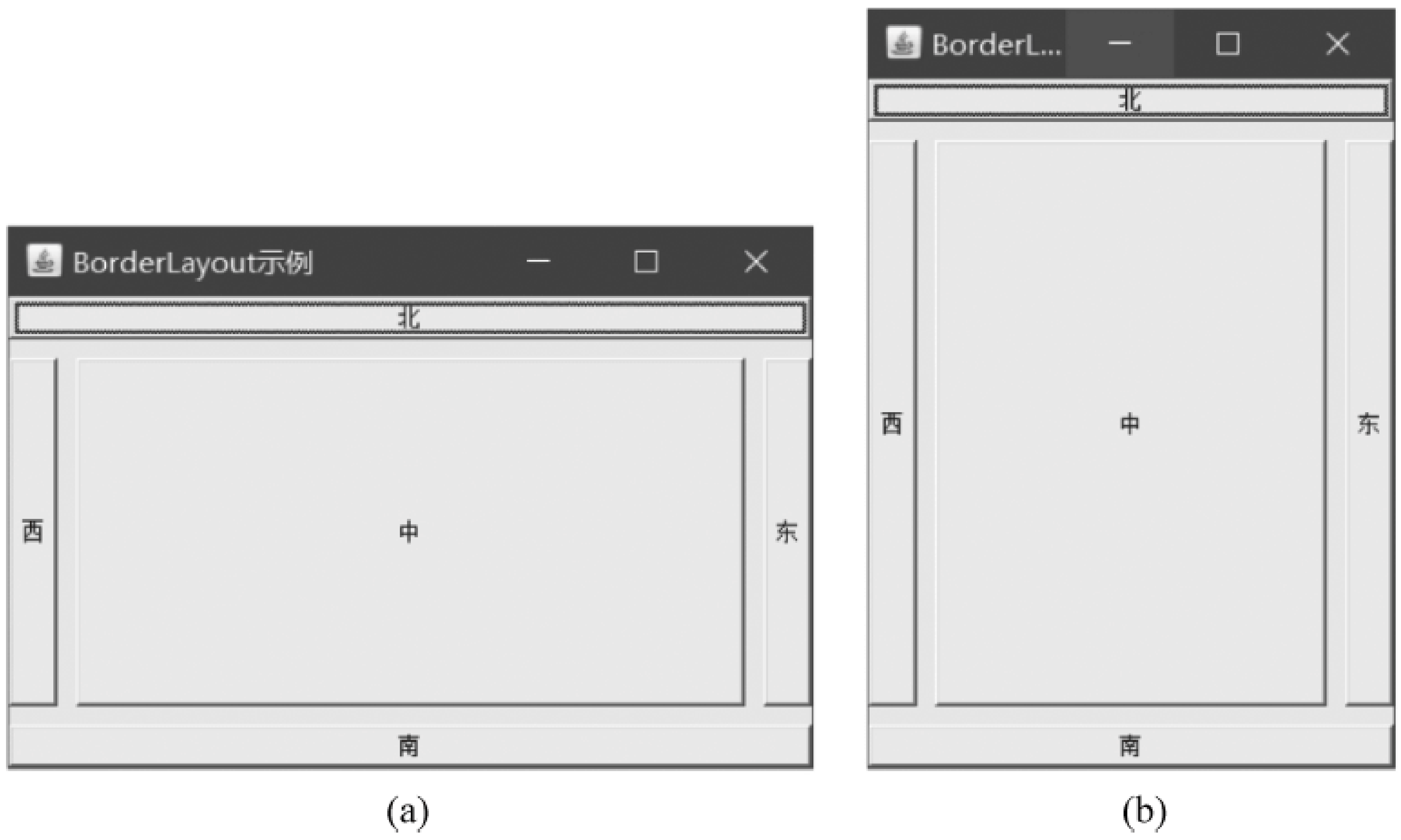


图 23-12 BorderLayout 布局与容器大小变化

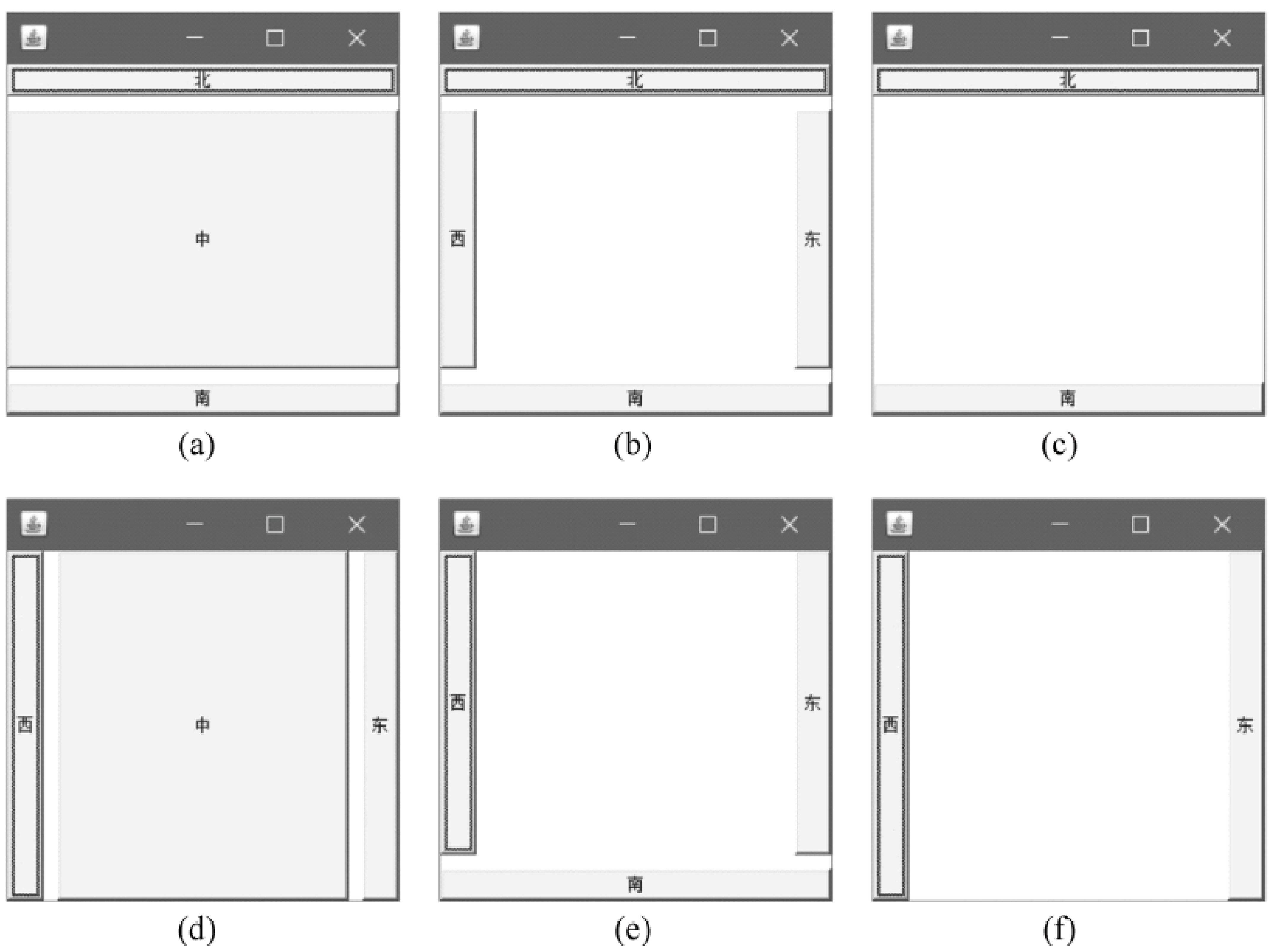


图 23-13 某个区域缺少组件

23.4.3 GridLayout 布局

GridLayout 布局以网格形式对组件进行摆放，容器被分成大小相等的矩形，一个矩形中放置一个组件。

GridLayout 布局主要的构造方法如下。

- GridLayout()：创建具有默认值的 GridLayout 对象，即每个组件占据一行一列。
- GridLayout(int rows, int cols)：创建具有指定行数和列数的 GridLayout 对象。
- GridLayout(int rows, int cols, int hgap, int vgap)：创建具有指定行数和列数的 GridLayout 对象，并指定水平和垂直间隙。

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.Button;
import java.awt.GridLayout;

import javax.swing.JFrame;

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);

        //设置 3 行 3 列的 GridLayout 布局管理器
        setLayout(new GridLayout(3, 3));                    ①

        //添加按钮到第一行的第一格
        getContentPane().add(new Button("1"));              ②
        //添加按钮到第一行的第二格
        getContentPane().add(new Button("2"));
        //添加按钮到第一行的第三格
        getContentPane().add(new Button("3"));
        //添加按钮到第二行的第一格
        getContentPane().add(new Button("4"));
        //添加按钮到第二行的第二格
        getContentPane().add(new Button("5"));
        //添加按钮到第二行的第三格
        getContentPane().add(new Button("6"));
        //添加按钮到第三行的第一格
        getContentPane().add(new Button("7"));
        //添加按钮到第三行的第二格
        getContentPane().add(new Button("8"));
        //添加按钮到第三行的第三格
        getContentPane().add(new Button("9"));              ③

        setSize(400, 400);
        setVisible(true);
    }
}
```

上述代码第①行设置当前窗口布局采用3行3列的GridLayout布局，它有9个区域，分别从左到右、从上到下摆放。代码第②～③行添加了9个Button，运行结果如图23-14所示。

GridLayout布局将容器分成几个区域，也会出现某个区域缺少组件的情况，GridLayout布局会根据行列划分的不同平均占据容器的空间，实际情况比较复杂。图23-15中列出了一些主要情况。

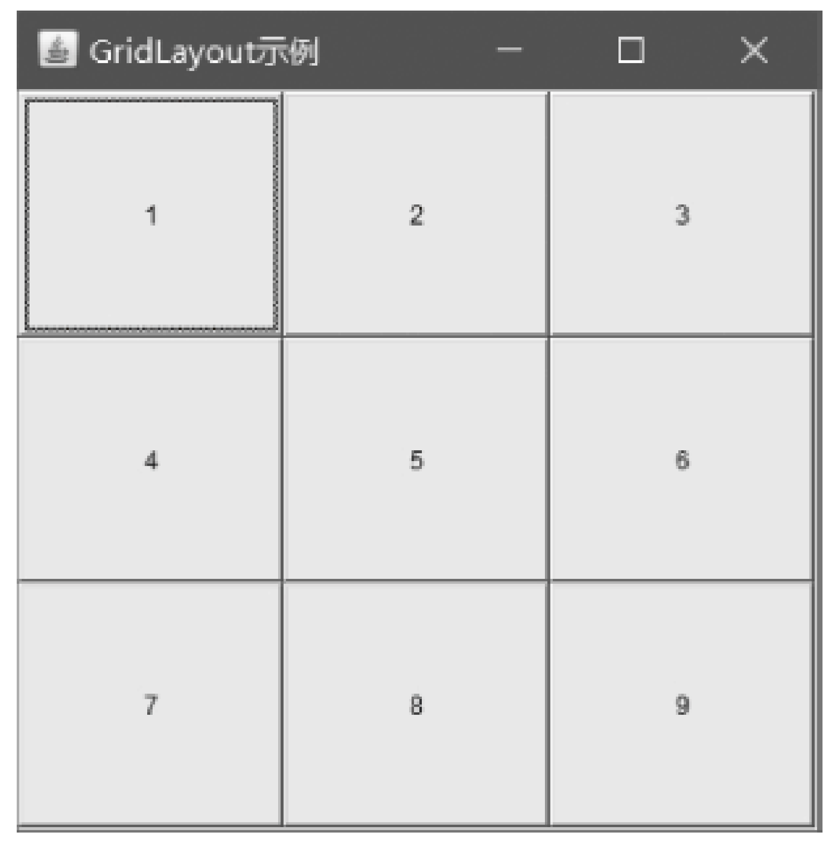
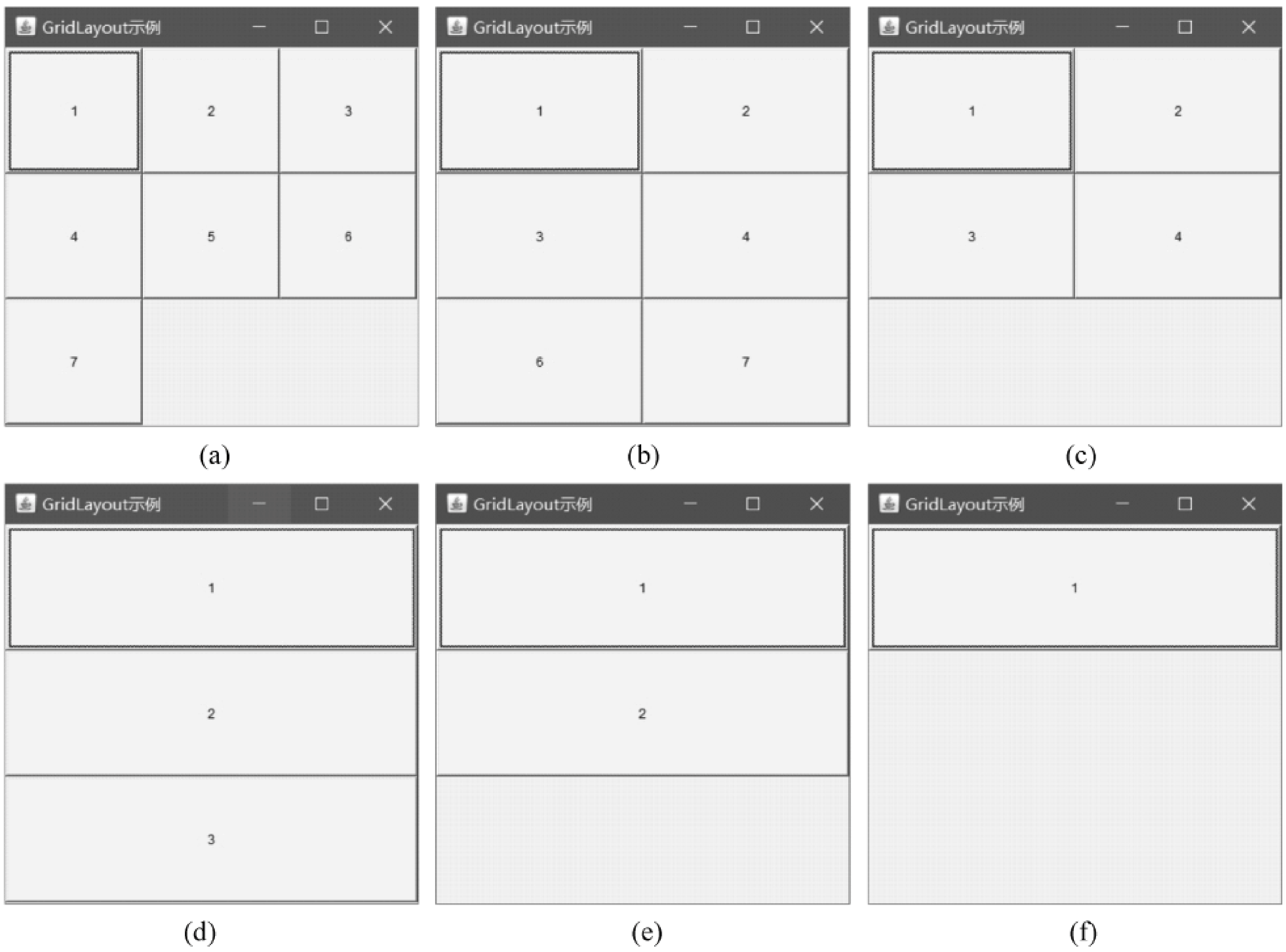


图23-14　GridLayout布局示例运行结果



(a)　(b)　(c)

(d)　(e)　(f)

图23-15　某个区域缺少组件

23.4.4　不使用布局管理器

如果要开发的图形用户界面应用不考虑跨平台，不考虑动态布局，窗口大小又不变，那么布局管理器就失去了使用的意义。容器也可以不设置布局管理器，那么此时的布局是由开发人员自己管理的。

组件有3个与布局有关的方法，即setLocation()、setSize()和setBounds()，在设置了布局管理的容器中组件的这几个方法不起作用，不设置布局管理时它们才起作用。

这3个方法的说明如下。

- void setLocation(int x, int y)：设置组件的位置。
- void setSize(int width, int height)：设置组件的大小。

- void setBounds(int x, int y, int width, int height)：设置组件的大小和位置。

下面通过示例介绍一下不使用布局管理器的情况，如图 23-16 所示。

示例代码如下：



图 23-16　不使用布局管理器示例

```
//MyFrame.java 文件
package com.a51work6;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);

        //设置窗口大小不变
        setResizable(false);                                              ①

        //不设置布局管理器
        getContentPane().setLayout(null);                                 ②

        //创建标签
        JLabel label = new JLabel("Label");
        //设置标签的位置和大小
        label.setBounds(89, 13, 100, 30);                                 ③
        //设置标签文本水平居中
        label.setHorizontalAlignment(SwingConstants.CENTER);              ④
        //添加标签到内容面板
        getContentPane().add(label);

        //创建 Button1
        JButton button1 = new JButton("Button1");
        //设置 Button1 的位置和大小
        button1.setBounds(89, 59, 100, 30);                               ⑤
        //添加 Button1 到内容面板
        getContentPane().add(button1);

        //创建 Button2
        JButton button2 = new JButton("Button2");
        //设置 Button2 的位置
        button2.setLocation(89, 102);                                     ⑥
        //设置 Button2 的大小
        button2.setSize(100, 30);                                         ⑦
        //添加 Button2 到内容面板
        getContentPane().add(button2);

        //设置窗口大小
        setSize(300, 200);
        //设置窗口可见
        setVisible(true);
```

```
        //注册事件监听器,监听 Button2 单击事件
        button2.addActionListener((event) -> {
            label.setText("Hello Swing!");
        });

        //注册事件监听器,监听 Button1 单击事件
        button1.addActionListener((event) -> {
            label.setText("Hello World!");
        });
    }
}
```

上述代码第①行设置不能调整窗口大小,没有设置布局管理器时,容器中的组件都绝对布局,容器大小如果变化,那么其中的组件大小和位置都不会变化。如图 23-17 所示,将窗口拉大后,组件还是在原来的位置。



图 23-17　不使用布局管理器后调整窗口大小

代码第②行的 setLayout(null)方法是不设置布局管理器,参数是 null。

代码第③行和第⑤行是通过调用 setBounds()方法设置组件的大小和位置,也可以分别调用 setSize()和 setLocation()方法设置组件的大小和位置,实现与 setBounds()方法相同的效果,见代码第⑥行和第⑦行。

另外,代码第④行的 setHorizontalAlignment(SwingConstants.CENTER)方法设置了标签的文本水平居中。

23.4.5　使用可视化设计工具

通过前面的学习,应该已经感受到:通过代码实现界面布局的工作量非常大。是否有可视化设计工具呢?各个主流的 Java IDE 工具都提供了可视化设计工具。IntelliJ IDEA 和 NetBeans IDE 都内置了可视化设计工具,如图 23-18 和图 23-19 所示。

Eclipse 本身不提供可视化工具,但是可以安装其他可视化设计工具插件实现,目前流行的是 WindowBuilder(http://www.eclipse.org/windowbuilder/),安装插件 WindowBuilder 的网址是 http://www.eclipse.org/windowbuilder/download.php,找到适合自己的 Eclipse 版本的在线安装地址,可以参考 2.2.2 节在线安装插件 WindowBuilder,安装过程这里不再赘述。WindowBuilder 插件界面如图 23-20 所示。

下面重点介绍 WindowBuilder 工具的使用。使用 WindowBuilder 工具可以创建一个

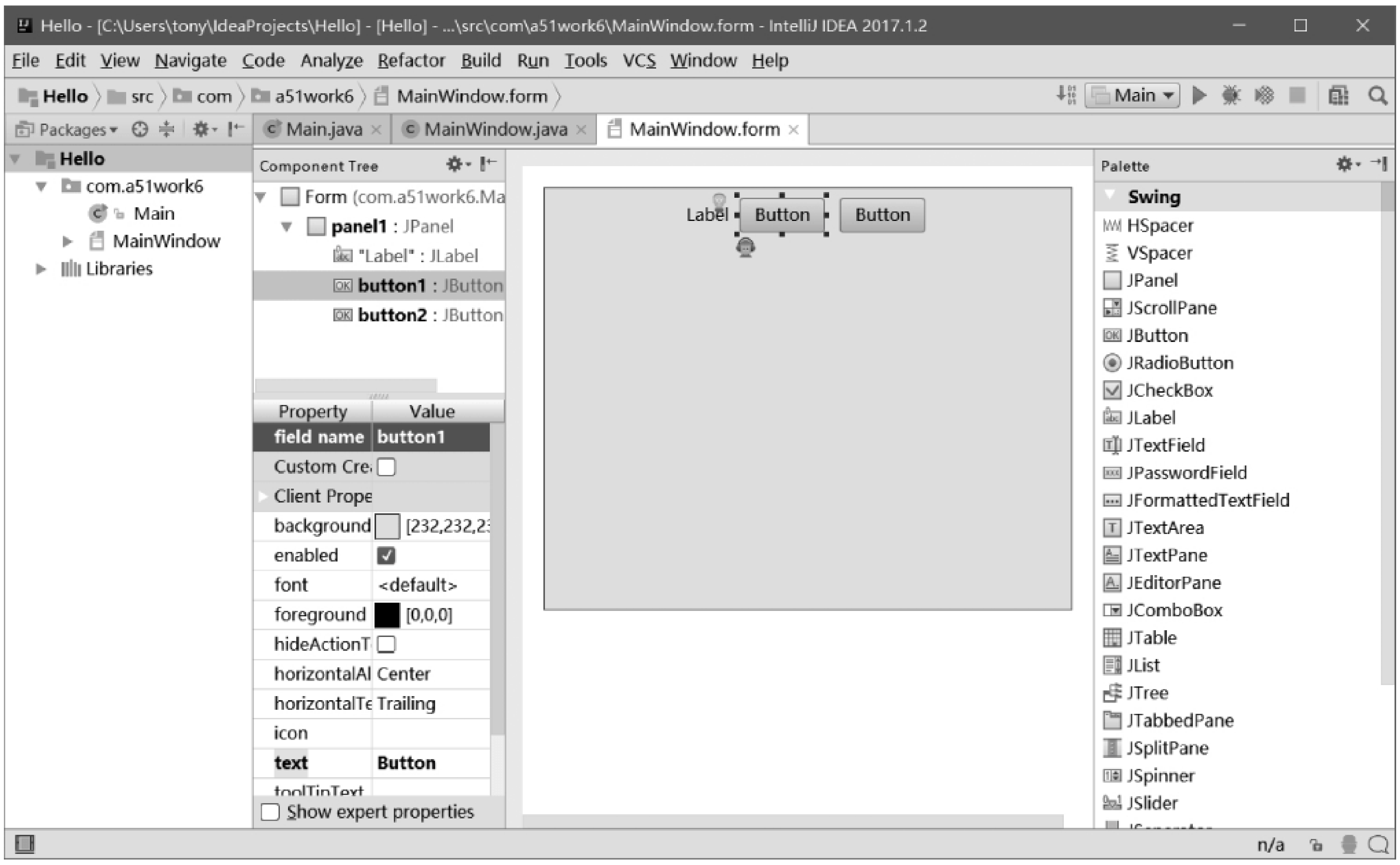


图 23-18　IntelliJ IDEA 可视化设计工具

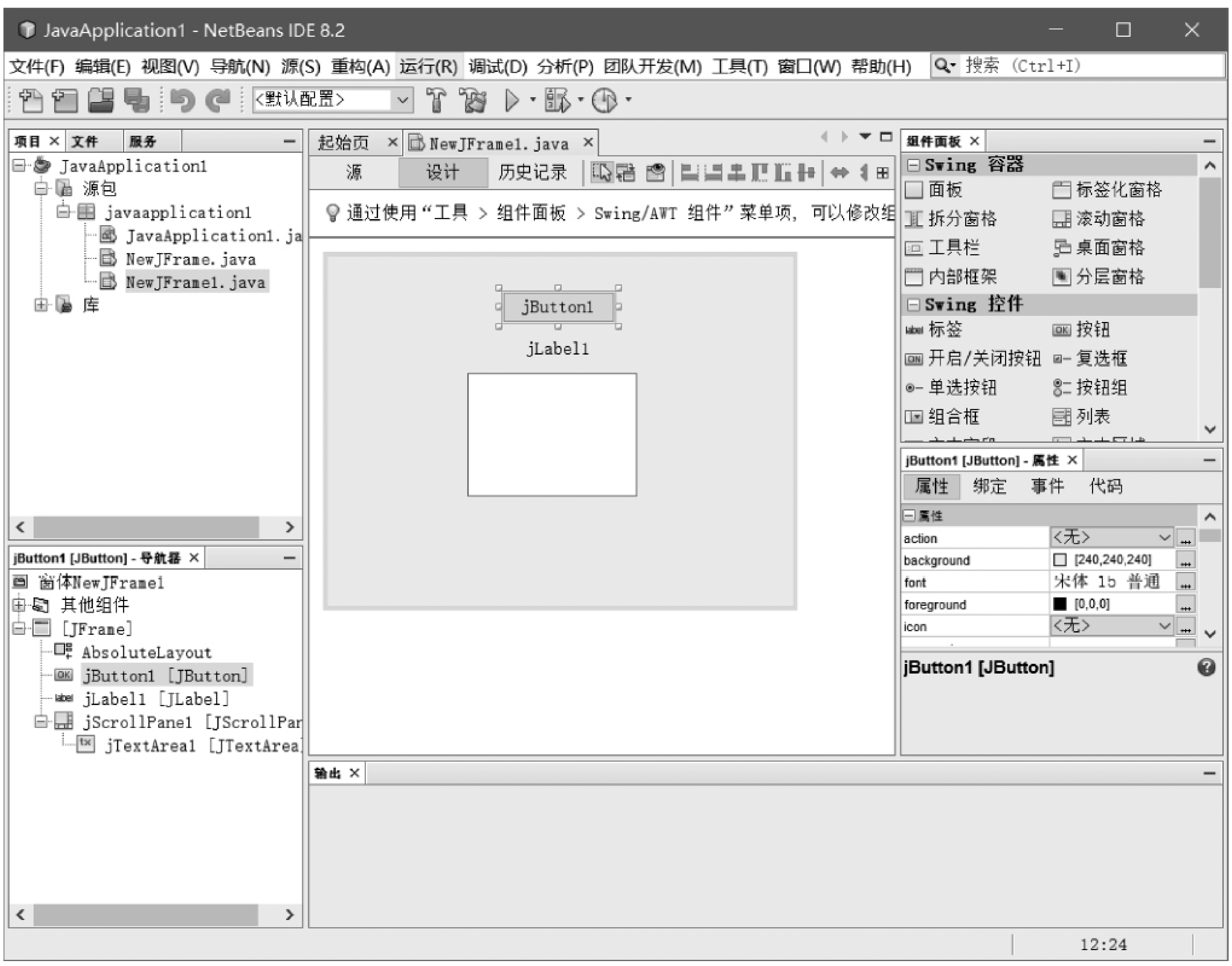


图 23-19　NetBeans IDE 可视化设计工具



图 23-20 WindowBuilder

新的窗口容器，也可以打开已经存在的窗口容器。

使用 WindowBuilder 工具创建新的窗口容器操作过程为：选择要放置窗口源代码的包，然后选择“文件”→“新建”→“其他”命令，打开新建类对话框，在对话框中找到文件夹 WindowBuilder→Swing Designer，如图 23-21 所示，选择其中的 JFrame，然后创建窗口类。

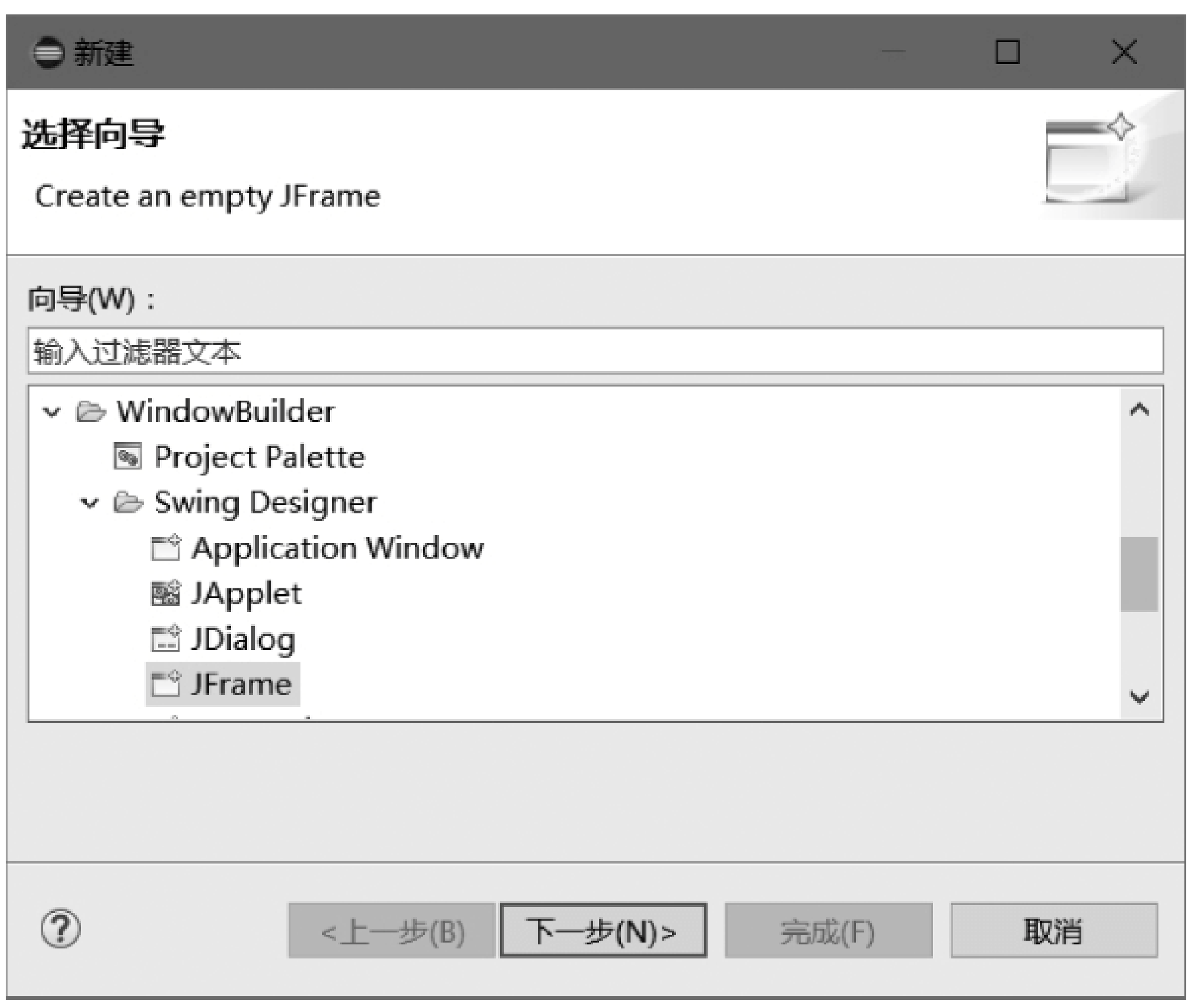


图 23-21 WindowBuilder 中创建窗口

如果已经编写了一个窗口类，则可以通过 WindowBuilder 打开。右击窗口源代码文件，在弹出的快捷菜单中选择“打开方式”→WindowBuilder Editor 命令即可。

这些可视化设计工具都是所见即所得的，使用起来比较简单，这里不再赘述。

23.5 Swing 组件

Swing 所有组件都继承自 JComponent，主要有文本处理、按钮、标签、列表、面板、组合框、滚动条、滚动面板、菜单、表格和树等组件。下面介绍一下常用的组件。

23.5.1 标签和按钮

标签和按钮在前面示例中已经用到了，本节再深入地介绍一下它们。

Swing 中标签类是 JLabel，它不仅可以显示文本还可以显示图标。JLabel 的构造方法如下。

- JLabel()：创建一个无图标无标题标签对象。
- JLabel(Icon image)：创建一个具有图标的标签对象。
- JLabel(Icon image, int horizontalAlignment)：通过指定图标和水平对齐方式创建标签对象。
- JLabel(String text)：创建一个标签对象，并指定显示的文本。
- JLabel(String text, Icon icon, int horizontalAlignment)：通过指定显示的文本、图标和水平对齐方式创建标签对象。
- JLabel(String text, int horizontalAlignment)：通过指定显示的文本和水平对齐方式创建标签对象。

上述构造方法 horizontalAlignment 参数是水平对齐方式，它的取值是 SwingConstants 中定义的以下常量之一：LEFT、CENTER、RIGHT、LEADING 或 TRAILING。

Swing 中的按钮类是 JButton，JButton 不仅可以显示文本还可以显示图标。JButton 常用的构造方法如下。

- JButton()：创建不带文本或图标的按钮对象。
- JButton(Icon icon)：创建一个带图标的按钮对象。
- JButton(String text)：创建一个带文本的按钮对象。
- JButton(String text, Icon icon)：创建一个带初始文本和图标的按钮对象。

下面通过示例介绍在标签和按钮中使用图标。如图 23-22 所示，界面中上面的图标是标签，下面的两个图标是按钮，当单击按钮时标签可以切换图标。



(a) (b)

图 23-22 标签和按钮示例

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;

import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

public class MyFrame extends JFrame {
    //用于标签切换的图标
    private static Icon images[] = { new ImageIcon("./icon/0.png"),
            new ImageIcon("./icon/1.png"),
            new ImageIcon("./icon/2.png"),
            new ImageIcon("./icon/3.png"),
            new ImageIcon("./icon/4.png"),
            new ImageIcon("./icon/5.png") };                                    ①

    //当前页索引
    private static int currentPage = 0;                                         ②

    public MyFrame(String title) {
        super(title);

        //设置窗口大小不变
        setResizable(false);

        //不设置布局管理器
        getContentPane().setLayout(null);                                       ③

        //创建标签
        JLabel label = new JLabel(images[0]);
        //设置标签的位置和大小
        label.setBounds(94, 27, 100, 50);
        //设置标签文本水平居中
        label.setHorizontalAlignment(SwingConstants.CENTER);
        //添加标签到内容面板
        getContentPane().add(label);

        //创建向后翻页按钮
        JButton backButton = new JButton(new ImageIcon("./icon/ic_menu_ back.png"));④
        //设置按钮的位置和大小
        backButton.setBounds(77, 90, 47, 30);
        //添加按钮到内容面板
        getContentPane().add(backButton);

        //创建向前翻页按钮
```

```
        JButton forwardButton = new JButton(new ImageIcon("./icon/ic_menu_forward.png"));
                                                                                     ⑤
        //设置按钮的位置和大小
        forwardButton.setBounds(179, 90, 47, 30);
        //添加按钮到内容面板
        getContentPane().add(forwardButton);

        //设置窗口大小
        setSize(300, 200);
        //设置窗口可见
        setVisible(true);

        //注册事件监听器,监听向后翻页按钮单击事件
        backButton.addActionListener((event) -> {
            if (currentPage < images.length - 1) {
                currentPage++;
            }
            label.setIcon(images[currentPage]);
        });

        //注册事件监听器,监听向前翻页按钮单击事件
        forwardButton.addActionListener((event) -> {
            if (currentPage > 0) {
                currentPage--;
            }
            label.setIcon(images[currentPage]);
        });

    }
}
```

上述代码第①行定义 ImageIcon 数组，用于标签切换图标，注意 Icon 是接口，ImageIcon 是实现 Icon 接口。代码第②行 currentPage 变量记录了当前页索引，前后翻页按钮会改变前页索引。

代码第③行是不设置布局管理器，代码第④行和第⑤行是创建向前和向后翻页按钮，构造方法参数是 ImageIcon 对象。

23.5.2 文本输入组件

文本输入组件主要有文本框（JTextField）、密码框（JPassword Field）和文本区（JTextArea）。文本框和密码框都只能输入和显示单行文本。当按下 Enter 键时，可以触发 ActionEvent 事件。而在文本区可以输入和显示多行多列文本。

文本框（JTextField）常用的构造方法如下。

- JTextField()：创建一个空的文本框对象。
- JTextField(int columns)：指定列数，创建一个空的文本框对象，列数是文本框显示的宽度，主要用于 FlowLayout 布局。
- JTextField(String text)：创建文本框对象，并指定初始化文本。

• JTextField(String text, int columns)：创建文本框对象，并指定初始化文本和列数。

JPasswordField 继承自 JTextField 构造方法类似，这里不再赘述。

文本区(JTextArea)常用的构造方法如下。

• JTextArea()：创建一个空的文本区对象。
• JTextArea(int rows, int columns)：创建文本区对象，并指定行数和列数。
• JTextArea(String text)：创建文本区对象，并指定初始化文本。
• JTextArea(String text, int rows, int columns)：创建文本区对象，并指定初始化文本、行数和列数。

下面通过示例介绍一下文本输入组件。如图 23-23 所示，界面中有三个标签(TextField:、Password:和 TextArea:)，一个文本框、一个密码框和一个文本区。这个布局有点复杂，可以采用布局嵌套，如图 23-24 所示，将 TextField: 标签、Password: 标签、文本框和密码框都放到一个面板(panel1)中；将 TextArea: 和文本区放到一个面板(panel2)中。两个面板 panel1 和 panel2 放到内容视图中，内容视图采用 BorderLayout 布局，每个面板内部采用 FlowLayout 布局。



图 23-23　文本输入组件示例

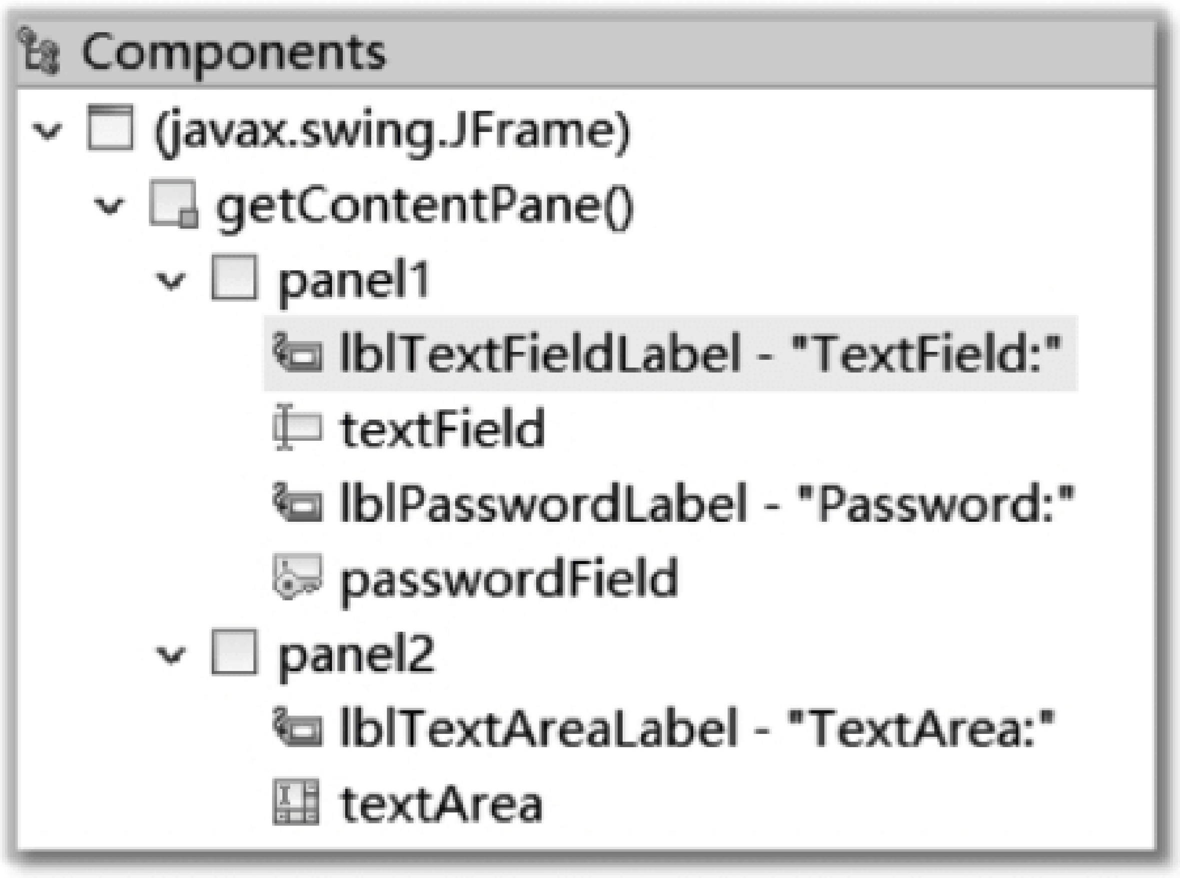


图 23-24　布局嵌套

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;

import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class MyFrame extends JFrame {
    private JTextField textField;
```

```
    private JPasswordField passwordField;

    public MyFrame(String title) {
        super(title);

        //设置布局管理 BorderLayout
        getContentPane().setLayout(new BorderLayout());

        //创建一个面板 panel1 放置 TextField 和 Password
        JPanel panel1 = new JPanel();                                ①
        //将面板 panel1 添加到内容视图
        getContentPane().add(panel1, BorderLayout.NORTH);            ②

        //创建标签
        JLabel lblTextFieldLabel = new JLabel("TextField:");
        //添加标签到面板 panel1
        panel1.add(lblTextFieldLabel);

        //创建文本框
        textField = new JTextField(12);                              ③
        //添加文本框到面板 panel1
        panel1.add(textField);


        //创建标签
        JLabel lblPasswordLabel = new JLabel("Password:");
        //添加标签面板 panel1
        panel1.add(lblPasswordLabel);

        //创建密码框
        passwordField = new JPasswordField(12);                      ④
        //添加密码框到面板 panel1
        panel1.add(passwordField);

        //创建一个面板 panel2 放置 TextArea
        JPanel panel2 = new JPanel();                                ⑤
        getContentPane().add(panel2, BorderLayout.SOUTH);            ⑥

        //创建标签
        JLabel lblTextAreaLabel = new JLabel("TextArea:");
        //添加标签到面板 panel2
        panel2.add(lblTextAreaLabel);

        //创建文本区
        JTextArea textArea = new JTextArea(3, 20);                   ⑦
        //添加文本区到面板 panel2
        panel2.add(textArea);

        //设置窗口大小
        pack();       //紧凑排列,其作用相当于 setSize()              ⑧
```

```
            //设置窗口可见
            setVisible(true);

            textField.addActionListener((event) ->{                  ⑨
                textArea.setText("在文本框上按下 Enter 键");
            });
        }
    }
```

上述代码第①行和第⑤行是创建面板容器，面板(JPanel)是一种没有标题栏和边框的容器，经常用于嵌套布局。然后将这两个面板添加到内容视图中，见代码第②行和第⑥行。

代码第③行创建文本框对象，指定列数是 12。代码第④行是创建密码框，指定列数是 12。它们都添加到面板 panel1 中。

代码第⑦行创建文本区对象，指定行数为 3，列数为 20，并将其添加到面板 panel2 中。

代码第⑧行 pack()设置窗口的大小，它设置的大小是将容器中所有组件刚好包裹进去。

代码第⑨行是文本框 textField 注册 ActionEvent 事件，当用户在文本框中按下 Enter 键时触发。

23.5.3 复选框和单选按钮

Swing 中提供了用于多选和单选功能的组件。

多选组件是复选框(JCheckBox)，复选框有时也单独使用，能提供两种状态的开和关。

单选组件是单选按钮(JRadioButton)，同一组的多个单选按钮应该具有互斥特性，这也是为什么单选按钮也叫作收音机按钮(RadioButton)，就是当一个按钮按下时，其他按钮一定抬起。同一组多个单选按钮应该放到同一个 ButtonGroup 对象，ButtonGroup 对象不属于容器，它会创建一个互斥作用范围。

JCheckBox 主要构造方法如下。

- JCheckBox()：创建一个没有文本、没有图标并且最初未被选定的复选框对象。
- JCheckBox(Icon icon)：创建有一个图标、最初未被选定的复选框对象。
- JCheckBox(Icon icon, boolean selected)：创建一个带图标的复选框对象，并指定其最初是否处于选定状态。
- JCheckBox(String text)：创建一个带文本的、最初未被选定的复选框对象。
- JCheckBox(String text, boolean selected)：创建一个带文本的复选框对象，并指定其最初是否处于选定状态。
- JCheckBox(String text, Icon icon)：创建带有指定文本和图标的、最初未被选定的复选框对象。
- JCheckBox(String text, Icon icon, boolean selected)：创建一个带文本和图标的复选框对象，并指定其最初是否处于选定状态。

JCheckBox 和 JRadioButton 有着相同的父类 JToggleButton、相同方法和类似的构造方法，因此 JRadioButton 构造方法不再赘述。

下面通过示例介绍一下复选框和单选按钮。如图 23-25 所示，界面中有一组复选框和一组单选按钮。



图 23-25　复选框和单选按钮示例

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;

…

public class MyFrame extends JFrame implements ItemListener {          ①

    //声明并创建 RadioButton 对象
    private JRadioButton radioButton1  = new JRadioButton("男");        ②
    private JRadioButton radioButton2  = new JRadioButton("女");        ③

    public MyFrame(String title) {
        super(title);

        //设置布局管理 BorderLayout
        getContentPane().setLayout(new BorderLayout());

        //创建一个面板 panel1 放置 TextField 和 Password
        JPanel panel1  = new JPanel();
        FlowLayout flowLayout_1  = (FlowLayout) panel1.getLayout();
        flowLayout_1.setAlignment(FlowLayout.LEFT);
        //将面板 panel1 添加到内容视图
        getContentPane().add(panel1, BorderLayout.NORTH);

        //创建标签
        JLabel lblTextFieldLabel  = new JLabel("选择你喜欢的编程语言：");
        //添加标签到面板 panel1
        panel1.add(lblTextFieldLabel);

        JCheckBox checkBox1  = new JCheckBox("Java");                 ④
        panel1.add(checkBox1);

        JCheckBox checkBox2  = new JCheckBox("C++");
        panel1.add(checkBox2);

        JCheckBox checkBox3  = new JCheckBox("Objective - C");
        //注册 checkBox3 对 ActionEvent 事件监听
        checkBox3.addActionListener((event)  -> {                      ⑤
            //打印 checkBox3 状态
            System.out.println(checkBox3.isSelected());
        });
        panel1.add(checkBox3);

        //创建一个面板 panel2 放置 TextArea
```

```
            JPanel panel2 = new JPanel();
            FlowLayout flowLayout = (FlowLayout) panel2.getLayout();
            flowLayout.setAlignment(FlowLayout.LEFT);
            getContentPane().add(panel2, BorderLayout.SOUTH);

            //创建标签
            JLabel lblTextAreaLabel = new JLabel("选择性别: ");
            //添加标签到面板 panel2
            panel2.add(lblTextAreaLabel);

            //创建 ButtonGroup 对象
            ButtonGroup buttonGroup = new ButtonGroup();                    ⑥
            //添加 RadioButton 到 ButtonGroup 对象
            buttonGroup.add(radioButton1);
            buttonGroup.add(radioButton2);

            //添加 RadioButton 到面板 panel2                                 ⑦
            panel2.add(radioButton1);
            panel2.add(radioButton2);

            //注册 ItemEvent 事件监听器
            radioButton1.addItemListener(this);                              ⑧
            radioButton2.addItemListener(this);

            //设置窗口大小
            pack();          //紧凑排列,其作用相当于 setSize()

            //设置窗口可见
            setVisible(true);
        }

        //实现 ItemListener 接口方法
        @Override
        public void itemStateChanged(ItemEvent e) {                         ⑨

            if (e.getStateChange() == ItemEvent.SELECTED) {                 ⑩
                JRadioButton button = (JRadioButton) e.getItem();
                System.out.println(button.getText());
            }
        }
    }
```

上述代码第②行和第③行创建了两个单选按钮对象,为了能让这两个单选按钮互斥,则需要把它们添加到一个 ButtonGroup 对象,见代码第⑥行的创建 ButtonGroup 对象,并把它们添加进来。为了监听两个单选按钮的选择状态,注册 ItemEvent 事件监听器,见代码第⑧行。为了一起处理两个单选按钮事件,它们需要使用同一个事件处理者,本例是 this,这说明当前窗口是事件处理者,它实现了 ItemListener 接口,见代码第①行。代码第⑨行实现了 ItemListener 接口的抽象方法。两个单选按钮使用同一个事件处理者,那么如何判断是

哪一个按钮触发的事件呢？代码第⑩行判断按钮是否被选中，如果选中，则通过 e.getItem()方法获得按钮引用，然后通过 getText()方法获得按钮的文本标签。

代码第④行创建了一个复选框对象，并且把它添加到面板 panel1 中。复选框和单选按钮都属于按钮，也能响应 ActionEvent 事件，代码第⑤行是注册 checkBox3 对 ActionEvent 事件监听。

23.5.4 下拉列表

Swing 中提供了下拉列表(JComboBox)组件，每次只能选择其中的一项。

JComboBox 常用的构造方法如下。

- JComboBox()：创建一个下拉列表对象。
- JComboBox(Object [] items)：创建一个下拉列表对象，items 设置下拉列表中的选项。下拉列表中的选项内容可以是任意类，而不再局限于 String。

下面通过示例介绍下拉列表组件。如图 23-26 所示，界面中有两个下拉列表组件。



(a) (b)

图 23-26 下拉列表示例

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;
…
public class MyFrame extends JFrame {

    //声明下拉列表 JComboBox
    private JComboBox choice1;
    private JComboBox choice2;

    private String[] s1 = { "Java", "C++", "Objective - C" };
    private String[] s2 = { "男", "女" };

    public MyFrame(String title) {
        super(title);

        getContentPane().setLayout(new GridLayout(2, 2, 0, 0));

        //创建标签
        JLabel lblTextFieldLabel = new JLabel("选择你喜欢的编程语言：");
        lblTextFieldLabel.setHorizontalAlignment(SwingConstants.RIGHT);
        getContentPane().add(lblTextFieldLabel);
```

```
            //实例化 JComboBox 对象
            choice1 = new JComboBox(s1);                        ①
            //注册 Action 事件侦听器,采用 Lambda 表达式
            choice1.addActionListener(e -> {                    ②
                JComboBox cb = (JComboBox) e.getSource();       ③
                //获得选择的项目
                String itemString = (String) cb.getSelectedItem();  ④
                System.out.println(itemString);
            });

            getContentPane().add(choice1);

            //创建标签
            JLabel lblTextAreaLabel = new JLabel("选择性别: ");
            lblTextAreaLabel.setHorizontalAlignment(SwingConstants.RIGHT);
            getContentPane().add(lblTextAreaLabel);

            //实例化 JComboBox 对象,采用 Lambda 表达式
            choice2 = new JComboBox(s2);                        ⑤
            //注册项目选择事件侦听器
            choice2.addItemListener(e -> {                      ⑥
                //项目选择
                if (e.getStateChange() == ItemEvent.SELECTED) { ⑦
                    //获得选择的项目
                    String itemString = (String) e.getItem();   ⑧
                    System.out.println(itemString);
                }
            });
            getContentPane().add(choice2);

            //设置窗口大小
            setSize(400, 150);

            //设置窗口可见
            setVisible(true);
        }
    }
```

上述代码第①行和第⑤行是创建下拉列表组件对象,其中构造方法参数是字符串数组。下拉列表组件在进行事件处理时,可以注册两种事件监听器:ActionListener 和 ItemListener,这两个监听器都只有一个抽象方法需要实现,因此可以采用 Lambda 表达式作为事件处理者,代码第②行和第⑥行分别注册这两个事件监听器。

代码第③行通过 e 事件参数获得事件源,代码第④行获得选中的项目,代码第⑦行判断当前的项目是否被选中,代码第⑧行从 e 事件参数中取出项目对象。

23.5.5 列表

Swing 中提供了列表(JList)组件,可以单选或多选。

JList 常用的构造方法如下。

- JList()：创建一个列表对象。
- JList(Object[] listData)：创建一个列表对象，listData 设置列表中的选项。列表中选项内容可以是任意类，而不再局限于 String。

下面通过示例介绍列表组件。如图 23-27 所示，界面中有一个列表组件。



图 23-27　列表示例

示例代码如下：

```
//MyFrame.java 文件
package com.a51work6;
…
public class MyFrame extends JFrame {

    private String[] s1 = { "Java", "C++", "Objective-C" };

    public MyFrame(String title) {
        super(title);
        //创建标签
        JLabel lblTextFieldLabel = new JLabel("选择你喜欢的编程语言：");
        getContentPane().add(lblTextFieldLabel, BorderLayout.NORTH);

        //列表组件 JList
        JList list1 = new JList(s1);                                      ①
        list1.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);      ②
        //注册项目选择事件侦听器,采用 Lambda 表达式
        list1.addListSelectionListener(e -> {                            ③
            if (e.getValueIsAdjusting() == false) {                       ④
                //获得选择的内容
                String itemString = (String) list1.getSelectedValue();    ⑤
                System.out.println(itemString);
            }
        });
        getContentPane().add(list1, BorderLayout.CENTER);

        //设置窗口大小
        setSize(300, 200);
        //设置窗口可见
        setVisible(true);
    }

}
```

上述代码第①行创建列表组件对象，代码第②行设置列表为单选，代码第③行选择列表事件，代码第④行中的 e.getValueIsAdjusting()==false 可以判断鼠标释放，e.getValueIsAdjusting()==true 可以判断鼠标按下。

代码第⑤行取出 getSelectedValue()获得选中的项目值，如果是多选，则可以通过 getSelectedValues()获得选中的项目值。

23.5.6 分隔面板

Swing 中提供了一种分隔面板(JSplitPane)组件,可以将屏幕分成左右或上下两部分。JSplitPane 常用的构造方法如下。



图 23-28 分隔面板示例

- JSplitPane(int newOrientation):创建一个分隔面板,参数 newOrientation 指定布局方向,newOrientation 取值是 JSplitPane.HORIZONTAL_SPLIT(水平)或 JSplitPane.VERTICAL_SPLIT(垂直)。
- JSplitPane(int newOrientation, Component newLeft Component, Component newRightComponent):创建一个分隔面板,参数 newOrientation 指定布局方向,newLeftComponent 指定左侧面板组件,newRight Component 指定右侧面板组件。

下面通过示例介绍分隔面板组件。如图 23-28 所示,界面分左右两部分,左边有列表组件,选中列表项目时右边会显示相应的图片。

示例代码如下:

```
//MyFrame.java 文件
package com.a51work6;
…
public class MyFrame extends JFrame {

    private String[] data = { "bird1.gif", "bird2.gif", "bird3.gif",
            "bird4.gif", "bird5.gif", "bird6.gif" };

    public MyFrame(String title) {
        super(title);

        //右边面板
        JPanel rightPane = new JPanel();
        rightPane.setLayout(new BorderLayout(0, 0));
        JLabel lblImage = new JLabel();
        lblImage.setHorizontalAlignment(SwingConstants.CENTER);
        rightPane.add(lblImage, BorderLayout.CENTER);

        //左边面板
        JPanel leftPane = new JPanel();
        leftPane.setLayout(new BorderLayout(0, 0));
        JLabel lblTextFieldLabel = new JLabel("选择鸟儿: ");
        leftPane.add(lblTextFieldLabel, BorderLayout.NORTH);

        //列表组件 JList
        JList list1 = new JList(data);
        list1.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        //注册项目选择事件侦听器,采用 Lambda 表达式
        list1.addListSelectionListener(e -> {
            if (e.getValueIsAdjusting() == false) {
```

```
                //获得选择的内容
                String itemString = (String) list1.getSelectedValue();
                String petImage = String.format("/images/%s", itemString);        ①
                Icon icon = new ImageIcon(MyFrame.class.getResource (petImage));  ②
                lblImage.setIcon(icon);
            }
        });
        leftPane.add(list1, BorderLayout.CENTER);

        //分隔面板
        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                    leftPane, rightPane);                        ③
        splitPane.setDividerLocation(100);                                       ④

        getContentPane().add(splitPane, BorderLayout.CENTER);                    ⑤

        //设置窗口大小
        setSize(300, 200);
        //设置窗口可见
        setVisible(true);
    }

}
```

上述代码分别创建两个面板，然后在代码第③行创建分隔面板，设置布局是水平方向和左右面板。代码第④行 splitPane.setDividerLocation(100)设置分隔条的位置，代码第⑤行将分隔面板添加到内容面板中。

代码第①行获得图片的相对路径，代码第②行创建图片 ImageIcon 对象，MyFrame.class.getResource(petImage)语句获取资源图片的绝对路径。

提示 资源文件是放在字节码文件夹中的文件，可通过 XXX.class.getResource()方法获得它运行时的绝对路径。

23.5.7 表格

当有大量数据需要展示时，可以使用二维表格，有时也可以使用表格修改数据。表格是非常重要的组件。Swing 提供了表格组件 JTable 类，但是表格组件比较复杂，它的表现形式与数据是分离的，Swing 的很多组件都是按照 MVC[①] 设计模式进行设计的，JTable 最有代表性，按照 MVC 设计理念 JTable 属于视图，对应的模型是 javax.swing.table.TableModel 接口实现类，根据自己的业务逻辑和数据实现 TableModel 接口。TableModel 接口要求实现所有抽象方法，使用起来比较麻烦，有时只是使用很简单的表格，此时可以使用 AbstractTableModel 抽象类。实际开发时需要继承 AbstractTableModel 抽象类。

JTable 类常用的构造方法如下。

① MVC 是一种设计理念，将一个应用分为模型(Model)、视图(View)和控制器(Controller)，它将业务逻辑、数据、界面表示进行分离的方法组织代码，界面表示的变化不会影响到业务逻辑组件，不需要重新编写业务逻辑。

- JTable(TableModel dm)：通过模型创建表格，dm 是模型对象，其中包含了表格要显示的数据。
- JTable(Object[][] rowData, Object[] columnNames)：通过二维数组和指定列名创建一个表格对象，rowData 是表格中的数据，columnNames 是列名。
- JTable(int numRows, int numColumns)：指定行和列数创建一个空的表格对象。

如图 23-29 所示为一个使用 JTable 表格示例。该表格放置在一个窗口中，由于数据比较多，还有滚动条。下面具体介绍一下如何通过 JTable 实现该示例。



书籍编号	书籍名称	作者	出版社	出版日期	库存数量
0036	高等数学	李放	人民邮电出版社	20000812	1
0004	FLASH精选	刘扬	中国纺织出版社	19990312	2
0026	软件工程	牛田	经济科学出版社	20000328	4
0015	人工智能	周末	机械工业出版社	19991223	3
0037	南方周末	邓光明	南方出版社	20000923	3
0008	新概念3	余智	外语出版社	19990723	2
0019	通讯与网络	欧阳杰	机械工业出版社	20000517	1
0014	期货分析	孙宝	飞鸟出版社	19991122	3
0023	经济概论	思佳	北京大学出版社	20000819	3
0017	计算机理论基础	戴家	机械工业出版社	20000218	4
0002	汇编语言	李利光	北京大学出版社	19980318	2
0033	模拟电路	邓英才	电子工业出版社	20000527	2
0011	南方旅游	王爱国	南方出版社	19990930	2
0039	黑幕	李仪	华光出版社	20000508	24

图 23-29　表格示例

先介绍通过二维数组和列名实现表格。这种方式创建表格不需要模型，实现起来比较简单。但是表格只能接受二维数组作为数据。

具体代码如下：

```
//MyFrameTable.java 文件
package com.a51work6.array;
…
public class MyFrameTable extends JFrame {

    //获得当前屏幕的宽和高
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize()
.getWidth();                                                          ①
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize()
.getHeight();                                                         ②

    private JTable table;
```

```
    public MyFrameTable(String title) {
        super(title);

        table = new JTable(rowData, columnNames);                          ③
        //设置表中内容字体
        table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
        //设置表列标题字体
        table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
        //设置表行高
        table.setRowHeight(40);
        //设置为单行选中模式
        table.setSelectionMode(javax.swing.ListSelectionModel.SINGLE_
SELECTION);
        //返回当前行的状态模型
        ListSelectionModel rowSM = table.getSelectionModel();
        //注册侦听器,选中行发生更改时触发
        rowSM.addListSelectionListener(new ListSelectionListener() {      ④

            public void valueChanged(ListSelectionEvent e) {
              //只处理鼠标按下
              if (e.getValueIsAdjusting() == false) {
                  return;
              }
                ListSelectionModel lsm = (ListSelectionModel) e.getSource();
                if (lsm.isSelectionEmpty()) {
                    System.out.println("没有选中行");
                } else {
                    int selectedRow = lsm.getMinSelectionIndex();
                    System.out.println("第" + selectedRow + "行被选中");
                }
            }
        });                                                                ⑤

        JScrollPane scrollPane = new JScrollPane();                        ⑥
        scrollPane.setViewportView(table);                                 ⑦
        getContentPane().add(scrollPane, BorderLayout.CENTER);

        //设置窗口大小
        setSize(960, 640);
        //计算窗口位于屏幕中心的坐标
        int x = (int) (screenWidth - 960) / 2;                            ⑧
        int y = (int) (screenHeight - 640) / 2;                           ⑨
        //设置窗口位于屏幕中心
        setLocation(x, y);

        //设置窗口可见
        setVisible(true);
    }

    //表格列标题
```

```
        String[] columnNames = { "书籍编号", "书籍名称", "作者", "出版社", "出版日期", "库存
    数量" };
        //表格数据
        Object[][] rowData = { { "0036", "高等数学", "李放", "人民邮电出版社", "20000812", 1 },
                { "0004", "Flash 精选", "刘扬", "中国纺织出版社", "19990312", 2 },
                { "0026", "软件工程", "牛田", "经济科学出版社", "20000328", 4 },
                { "0015", "人工智能", "周未", "机械工业出版社", "19991223", 3 },
                { "0037", "南方周末", "邓光明", "南方出版社", "20000923", 3 },
                                        ⋮
                { "0032", "SOL 使用手册", "贺民", "电子工业出版社", "19990425", 2 } };

    }
```

上述代码第①行和第②行获得当前机器屏幕的高和宽,通过屏幕高和宽可以计算出当前窗口屏幕居中时的坐标。代码第⑧行和第⑨行计算这个坐标,由于坐标原点在屏幕的左上角,所以窗口居中坐标公式:

```
    x = (屏幕宽度 - 窗口宽度) / 2
    y = (屏幕高度 - 窗口高度) / 2
```

代码第③行创建 JTable 表格对象,采用了二维数组和字符串一维数组创建表格对象。代码第④～⑤行注册事件监听器,监听器当行选择变化时触发。由于 List SelectionListener 接口虽然不是函数式接口,但只有一个方法,所以可以使用 Lambda 表达式实现该接口。修改代码如下:

```
    //也可换成 Lambda 表达式
    rowSM.addListSelectionListener(e -> {
        ListSelectionModel lsm = (ListSelectionModel) e.getSource();
        if (lsm.isSelectionEmpty()) {
            System.out.println("没有选中行");
        } else {
            int selectedRow = lsm.getMinSelectionIndex();
            System.out.println("第" + selectedRow + "行被选中");
        }
    });
```

表格一般都会放到一个滚动面板(JScrollPane)中,这可以保证数据超出屏幕时能够出现滚动条。把表格添加到滚动面板并不是使用 add()方法,而是使用代码第⑦行的 scrollPane.setViewportView(table)语句。滚动面板是非常特殊的面板,它管理着一个视口或窗口,当里面的内容超出视口(或窗口)时会出现滚动条,setViewportView()方法可以设置一个容器或组件作为滚动面板的视口。

23.6 案例:图书库存

在实际项目开发中往往数据是从数据库中查询返回的,数据结构有多种形式,采用自定义模型可以接收任何形式的数据。本节将 23.5 节的图书表格示例采用自定义模型进行

重构。

在进行数据库设计时,数据库中每一个表对应 Java 一个实体类,实体类是系统的"人""事""物"等一些名词,例如图书(Book)就是一个实体类。实体类 Book 代码如下:

```
//Book.java
package com.a51work6.entity;

//图书实体类
public class Book {

    //图书编号
    private String bookid;
    //图书名称
    private String bookname;
    //图书作者
    private String author;
    //出版社
    private String publisher;
    //出版日期
    private String pubtime;
    //库存数量
    private int inventory;

    public String getBookid() {
        return bookid;
    }
    public void setBookid(String bookid) {
        this.bookid = bookid;
    }
    public String getBookname() {
        return bookname;
    }
    public void setBookname(String bookname) {
        this.bookname = bookname;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    …
    //省略 Getter 和 Setter 方法
}
```

从代码可见,实体类有很多私有属性(成员变量),为了在类外部能够访问它们,一般都会提供公有的 Getter 和 Setter 方法。

提示 主流的 Java IDE 编程工具都提供了通过属性生成对应的 Getter 和 Setter 方法功能。所以一般程序员只编写属性,然后通过 IDE 工具生成。Eclipse 中生成 Getter 和

Setter的方法是：打开源代码文件，选择“源码”→“生成 Getter 和 Setter”命令，弹出如图23-30所示的对话框，根据需要选中相应的属性，也可以只生成 Getter 或 Setter 方法。选择完成之后单击“确定”按钮即可。



图 23-30 Eclipse 中生成 Getter 和 Setter 方法

由于目前没有介绍数据库编程，本例表格中的数据是从 JSON 文件 Books.json 中读取的，Books.json 位于项目的 db 目录中。JSON 文件 Books.json 的内容如下：

```
[{"bookid":"0036","bookname":"高等数学","author":"李放","publisher":"人民邮电出版社",
"pubtime":"20000812","inventory":1},
{"bookid":"0004","bookname":"FLASH 精选","author":"刘扬","publisher":"中国纺织出版社",
"pubtime":"19990312","inventory":2},
…
{"bookid":"0005","bookname":"java 基础","author":"王一","publisher":"电子工业出版社",
"pubtime":"19990528","inventory":3},
{"bookid":"0032","bookname":"SQL 使用手册","author":"贺民","publisher":"电子工业出版
社","pubtime":"19990425","inventory":2}]
```

从文件 Books.json 可见，整个文档结构是 JSON 数组，因为 JSON 字符串的开始和结尾被中括号括起来，这说明是 JSON 数组。JSON 数组的每一个元素是 JSON 对象，因为 JSON 对象是用大括号括起来的。代码如下：

```
{"bookid":"0032","bookname":"SQL 使用手册","author":"贺民","publisher":"电子工业出版
社","pubtime":"19990425","inventory":2}
```

清楚这个 JSON 文档结构非常必要，当编程时会根据这个文档结构解析 JSON 文档代码。参考 HelloWorld 代码如下：

```java
//HelloWorld.java 文件
package com.a51work6;
…

public class HelloWorld {

    public static void main(String[] args) {
        List<Book> data = readData();
        new MyFrameTable("图书库存", data);
    }

    //从文件中读取数据
    private static List<Book> readData() {
        //返回的数据列表
        List<Book> list = new ArrayList<Book>();
        //数据文件
        String dbFile = "./db/Books.json";

        try (FileInputStream fis = new FileInputStream(dbFile);
                InputStreamReader ir = new InputStreamReader(fis);
                BufferedReader in = new BufferedReader(ir)) {

            //1.读取文件
            StringBuilder sbuilder = new StringBuilder();
            String line = in.readLine();

            while (line != null) {
                sbuilder.append(line);
                line = in.readLine();
            }

            //2.JSON 解码
            //读取 JSON 字符完成
            System.out.println("读取 JSON 字符完成…");
            //JSON 解码,解码成功返回 JSON 数组
            JSONArray jsonArray = new JSONArray(sbuilder.toString());
            System.out.println("JSON 解码成功完成…");

            //3.将 JSON 数组放到 List<Book>集合中
            //遍历集合
            for (Object item : jsonArray) {

                JSONObject row = (JSONObject) item;

                Book book = new Book();
                book.setBookid((String) row.get("bookid"));
                book.setBookname((String) row.get("bookname"));
                book.setAuthor((String) row.get("author"));
                book.setPublisher((String) row.get("publisher"));
```

```
                book.setPubtime((String) row.get("pubtime"));
                book.setInventory((Integer) row.get("inventory"));

                list.add(book);
            }

        } catch (Exception e) {
        }

        return list;
    }
}
```

上述代码处理过程经历了以下 3 个步骤。

(1) 读取文件。通过 Java IO 取得文件./db/Books.json，每次读取的字符串保存到 StringBuilder 的 sbuilder 对象中。文件读完，sbuilder 中就是全部的 JSON 字符串。

(2) JSON 解码。读取 JSON 字符完成后，需要对其进行解码。由于 JSON 字符串是数组结构，因此解码时使用 JSONArray，创建 JSONArray 对象过程就是对字符串进行解码的过程，如果没有发生异常，则说明成功解码。

(3) 将 JSON 数组放到 List<Book>集合中。本例表格使用的数据格式不是 JSON 数组形式，而是 List<Book>，这种结构就是 List 集合中每一个元素都是 Book 类型。这个过程需要遍历 JSON 数组，把数据重新组装到 Book 对象中。

模型 BookTableModel 代码如下：

```
//BookTableModel.java 文件
package com.a51work6;

import java.util.List;

import javax.swing.table.AbstractTableModel;

public class BookTableModel extends AbstractTableModel {                    ①

    //列名数组
    private String[] columnNames = { "书籍编号", "书籍名称", "作者", "出版
社", "出版日期", "库存数量" };

    //data 保存了表格中数据,data 类型是 List 集合
    private List<Book> data = null;

    public BookTableModel(List<Book> data) {
        this.data = data;
    }

    //获得列数
    @Override
    public int getColumnCount() {
```

```
        return columnNames.length;
    }

    //获得行数
    @Override
    public int getRowCount() {
        return data.size();
    }

    //获得某行某列的数据
    @Override
    public Object getValueAt(int row, int col) {                    ②

        Book book = (Book) data.get(row);
        switch (col) {
        case 0:
            return book.getBookid();
        case 1:
            return book.getBookname();
        case 2:
            return book.getAuthor();
        case 3:
            return book.getPublisher();
        case 4:
            return book.getPubtime();
        case 5:
            return new Integer(book.getInventory());
        }
        return null;
    }

    //获得某列的名字
    @Override
    public String getColumnName(int col) {                          ③
        return columnNames[col];
    }
}
```

上述代码是自定义的模型，它继承了抽象类 AbstractTableModel，见代码第①行。抽象类 AbstractTableModel 要求必须实现 getColumnCount()、getRowCount()和 getValueAt() 3 个抽象方法，getColumnCount()方法提供表格列数，getRowCount()方法提供表格行数，getValueAt()方法提供了指定行和列时单元格内容。代码第③行的 getColumnName()方法不是抽象类要求实现的方法，重写该方法能够给表格提供有意义的列名。

窗口代码如下：

```
//MyFrameTable.java 文件
package com.a51work6;
…
```

```
public class MyFrameTable extends JFrame {

    //获得当前屏幕的宽和高
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();

    private JTable table;
    //图书列表
    private List<Book> data;

    public MyFrameTable(String title, List<Book> data) {
        super(title);

        this.data = data;
        TableModel model = new BookTableModel(data);

        table = new JTable(model);
        //设置表中内容字体
        table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
        //设置表列标题字体
        table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
        //设置表行高
        table.setRowHeight(40);
        //设置为单行选中模式
        table.setSelectionMode(javax.swing.ListSelectionModel.SINGLE_SELECTION);
        //返回当前行的状态模型
        ListSelectionModel rowSM = table.getSelectionModel();
        //注册侦听器,选中行发生更改时触发
        rowSM.addListSelectionListener(e -> {
            //只处理鼠标按下
            if (e.getValueIsAdjusting() == false) {
                return;
            }
            ListSelectionModel lsm = (ListSelectionModel) e.getSource();
            if (lsm.isSelectionEmpty()) {
                System.out.println("没有选中行");
            } else {
                int selectedRow = lsm.getMinSelectionIndex();
                System.out.println("第" + selectedRow + "行被选中");
            }
        });

        JScrollPane scrollPane = new JScrollPane();
        scrollPane.setViewportView(table);
        getContentPane().add(scrollPane, BorderLayout.CENTER);

        //设置窗口大小
        setSize(960, 640);
        //计算窗口位于屏幕中心的坐标
        int x = (int) (screenWidth - 960) / 2;
```

```
        int y = (int) (screenHeight - 640) / 2;
        //设置窗口位于屏幕中心
        setLocation(x, y);

        //设置窗口可见
        setVisible(true);
    }

}
```

窗口代码与23.5.7节的类似，这里不再赘述。

本章小结

本章介绍了Java中图形用户界面编程技术Swing，Swing的基础是AWT，Swing的事件处理和布局管理都是依赖于AWT，但是AWT提供的组件在使用开发中很少使用，因此重点学习Swing提供的组件。

23.7 同步练习

1. 下列哪些接口在Java中没有定义相对应的Adapter类？（　　）
   A. MouseListener　　B. KeyListener
   C. ActionListener　　D. ItemListener
   E. WindowListener
2. 下列选项中哪些是Java布局管理器类？（　　）
   A. FlowLayout　　B. BorderLayout
   C. GridLayout　　D. AbstractLayout
3. 下列哪些Java组件为容器组件？（　　）
   A. 下拉列表框　　B. 列表框
   C. 面板　　D. 按钮
4. 容器被重新设置大小后，哪种布局管理器容器中的组件大小不随容器大小的变化而改变？（　　）
   A. 绝对布局管理器　　B. FlowLayout
   C. BorderLayout　　D. GridLayout

附录 A 数据库编程

APPENDIX A

数据必须以某种方式来存储才可以调用。数据库实际上是一组相关数据的集合。例如，某个医疗机构中所有信息的集合可以被称为一个“医疗机构数据库”，这个数据库中的所有数据都与医疗机构相关。

与数据库编程相关的技术很多，涉及具体的数据库安装、配置和管理，还要掌握 SQL，最后才能编写程序访问数据库。本附录重点介绍 MySQL 数据库的安装和配置，以及 JDBC 数据库编程。

A.1 数据持久技术简介

把数据保存到数据库中只是一种数据持久化方式。凡是将数据保存到存储介质中，需要的时候能够找到它们，并能够对数据进行修改，这些就属于数据持久化。

Java 中数据持久化技术有很多。

1. 文本文件

通过 Java I/O 流技术将数据保存到文本文件中，然后进行读写操作，这些文件一般是结构化的文档，如 XML、JSON 和 CSV 等文件。结构化文档就是文件内部采取某种方式将数据组织起来。

2. 对象序列化

序列化用于将某个对象以及它的状态写到文件中，它保证了被写入的对象之间的关系，当需要这个对象时，可以完整地从文件重新构造出来，并保持原来的状态。在 Java 中实现 java.io.Serilizable 接口的对象才能被序列化和反序列化。Java 还提供了两个流：ObjectInputStream 和 ObjectOutputStream。但序列化不支持事务处理、查询或者向不同的用户共享数据。序列化只适用于最简单的应用，或者在某些无法有效地支持数据库的嵌入式系统中。

3. 数据库

将数据保存到数据库中是一个不错的选择，数据库的后面是一个数据库管理系统，它支持事务处理、并发访问、高级查询和 SQL。Java 对象保存到数据库中主要的技术有 JDBC[①]、EJB[②] 和 ORM[③] 框架等。JDBC 是本书重点介绍的技术。

---

① Java 数据库连接(Java DataBase Connectivity，JDBC)。

② 企业级 JavaBean(Enterprise JavaBean，EJB)是一个用来构筑企业级应用的服务器端组件。

③ 对象关系映射(Object-Relational Mapping，ORM)能将对象保存到数据库表中，对象与数据库表结构之间是有某种对应关系的。

A.2　MySQL 数据库管理系统

介绍 JDBC 技术一定会依托某个数据库管理系统(Database Management System，DBMS)，还会使用 SQL 语句，所以本节先介绍一下数据库管理系统。

数据库管理系统负责对数据进行管理、维护和使用。现在主流数据库管理系统有 Oracle、SQL Server、DB2、Sysbase 和 MySQL 等，本节介绍 MySQL 数据库管理系统的使用和管理。

MySQL(https://www.mysql.com)是流行的开放源码 SQL 数据库管理系统，它是由 MySQL AB 公司开发，先被 Sun 公司收购，后来又被 Oracle 公司收购，现在 MySQL 数据库是 Oracle 公司旗下的数据库产品，Oracle 公司负责提供技术支持和维护。

A.2.1　数据库的安装与配置

目前 Oracle 公司提供了多个 MySQL 版本，其中社区版 MySQL Community Edition 是免费的，比较适合中小企业。本书也采用这个版本介绍。

社区版下载地址是 https://dev.mysql.com/downloads/windows/installer/5.7.html，如图 A-1 所示，可以选择不同的平台版本，MySQL 可在 Windows、Linux 和 UNIX 等操作系统上安装和运行。本书选择的是 Windows 版中的 mysql-installer-community-5.7.18.1.msi 安装文件。

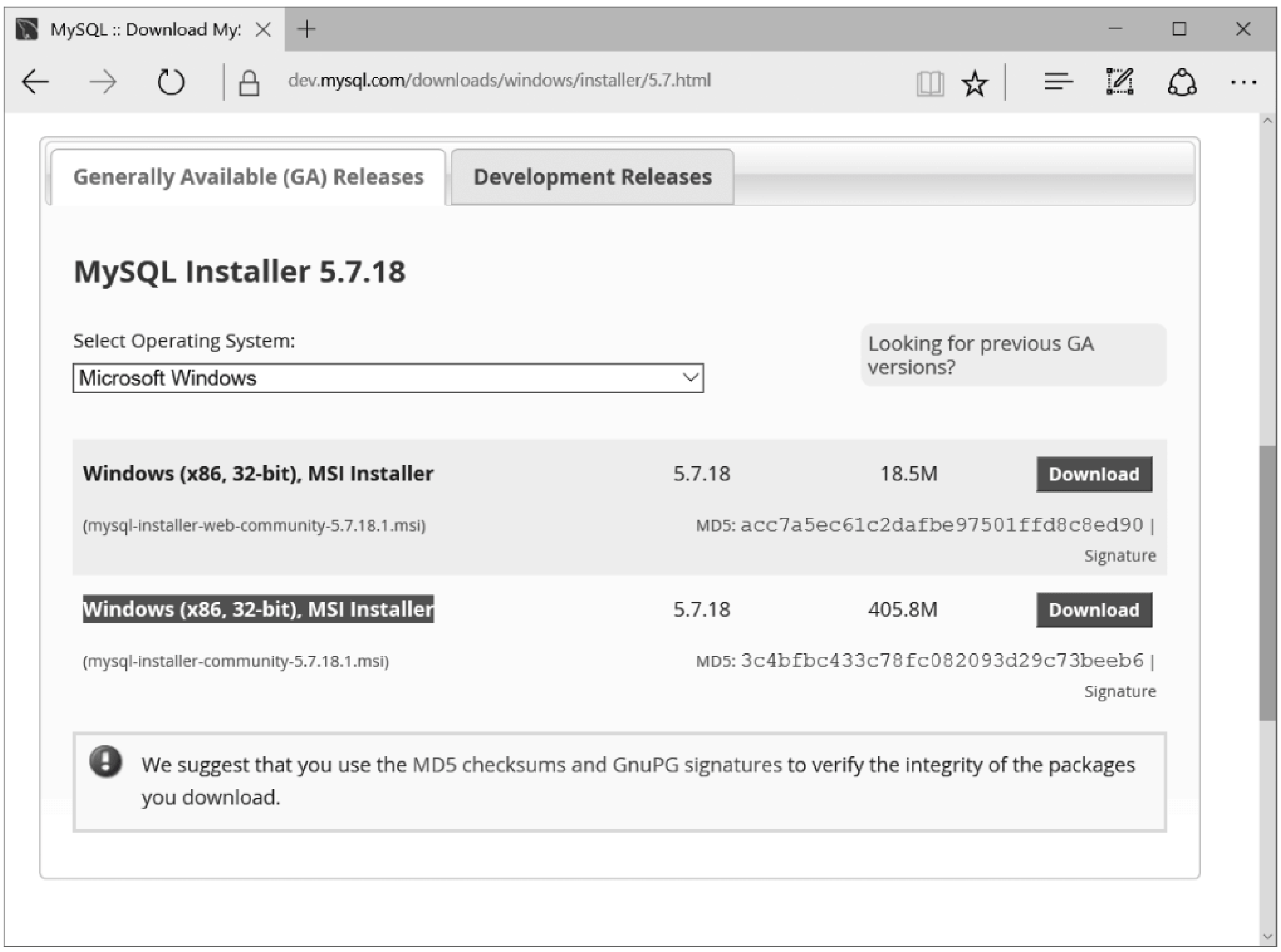
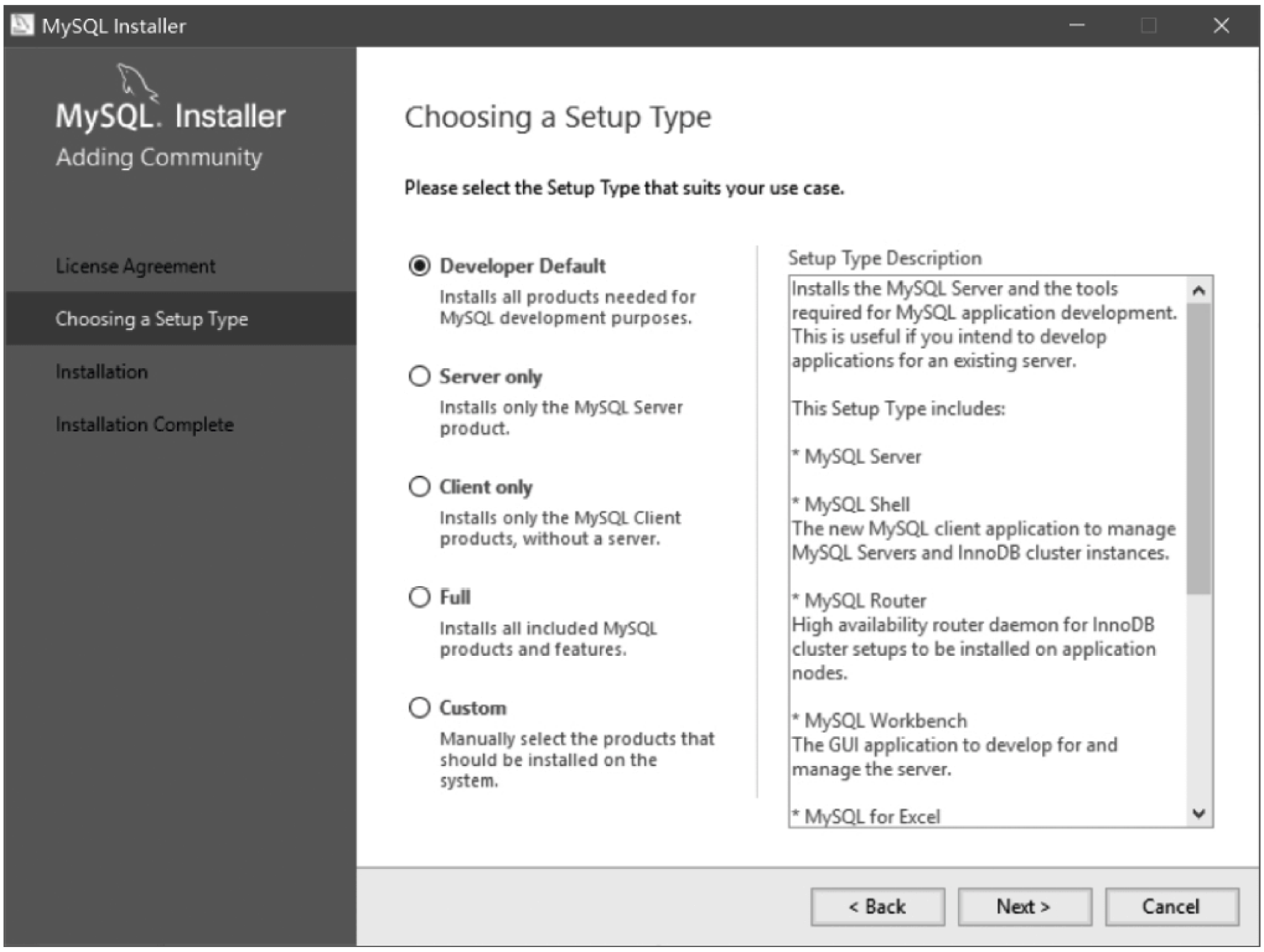


图 A-1　MySQL 数据库社区版下载

下载成功后，可以双击.msi文件启动安装过程。安装过程比较简单，这里介绍一个关键步骤。

1. 安装类型选择

如图A-2所示是安装类型选择对话框。在这个界面中可以选择安装类型，有5种安装类型：Developer Default（开发者安装）、Server only（只安装服务器）、Client only（只安装客户端）、Full（全部安装）和Custom（自定义安装）。对于学习和开发可以选择Developer Default安装。



图A-2　安装类型选择

2. 安装环境检查

在Windows下安装时，由于Windows版本的多样性，安装过程会检查你的需要，缺少哪些Windows安装包，安装过程会给出提示。如图A-3所示，安装MySQL Server需要Microsoft Visual C++ 2013 Runtime，这时需要到微软网站下载Microsoft Visual C++ 2013 Runtime安装包，安装好Microsoft Visual C++ 2013 Runtime后，再重新安装MySQL。

3. 配置过程

所需要的文件安装完成后就会进入MySQL的配置过程。如图A-4所示是数据库类型选择对话框，Standalone MySQL Server/Classic MySQL Replication是单个服务器，InnoDB Cluster Sandbox Test Setup(for testing only)是数据库集群。

在图A-4所示的对话框中选择Standalone MySQL Server / Classic MySQL Replication单选按钮，单击Next按钮进入如图A-5所示的服务器配置类型选择对话框。在这里可以选择配置类型、通信协议和端口等。单击Config Type下拉列表可以选择如下的配置类型。

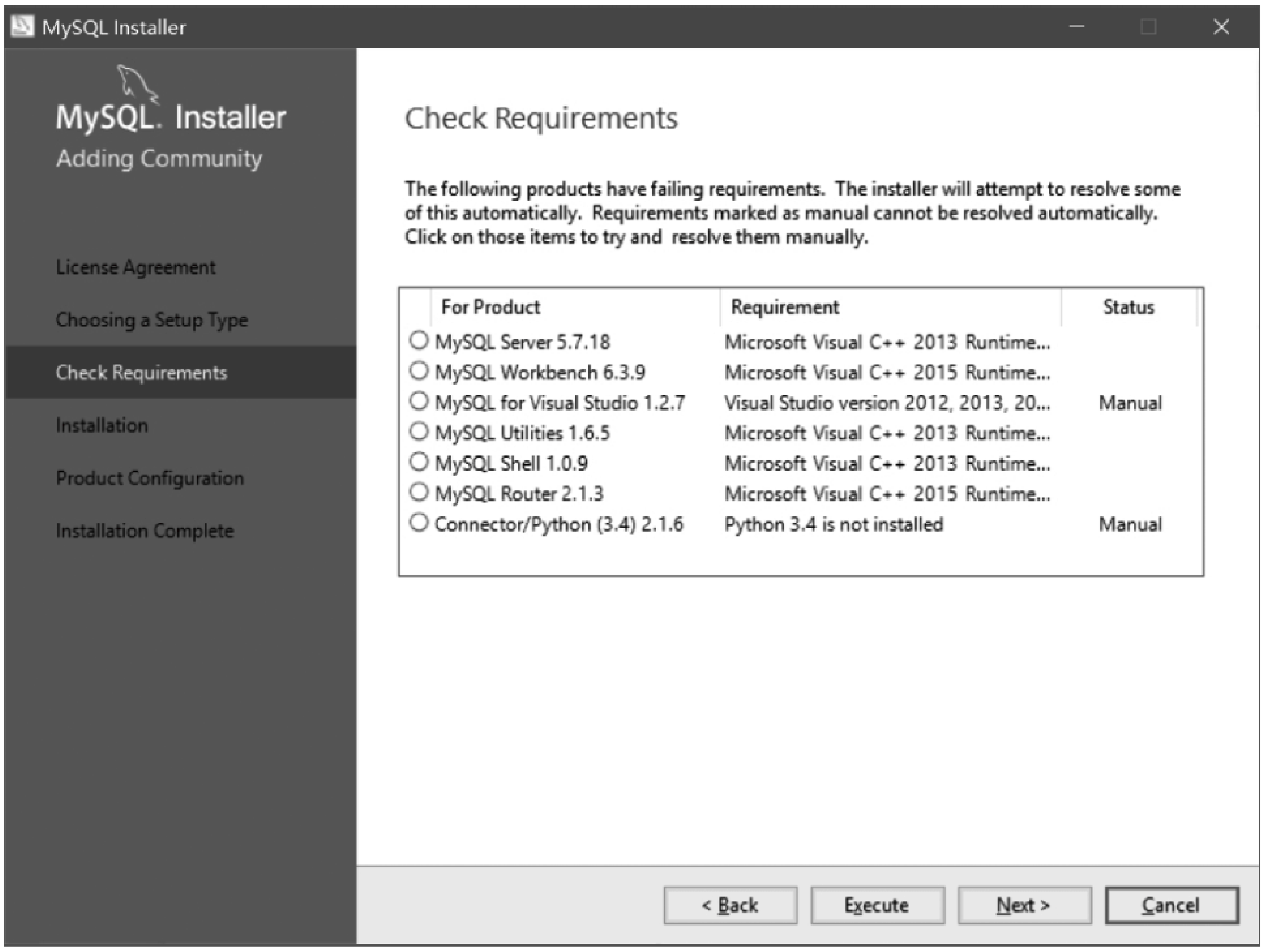


图 A-3 安装环境检查

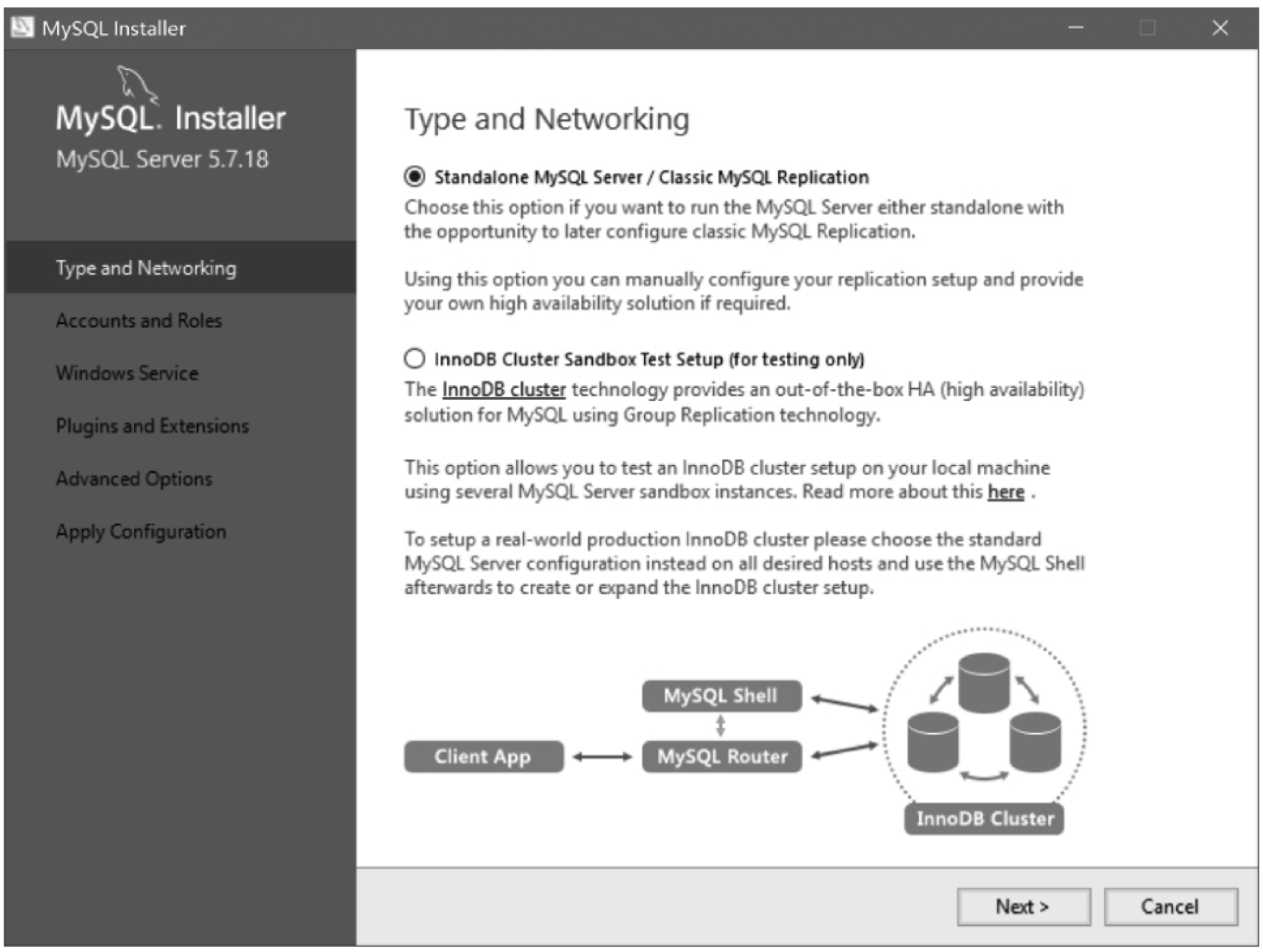


图 A-4 数据库类型选择对话框

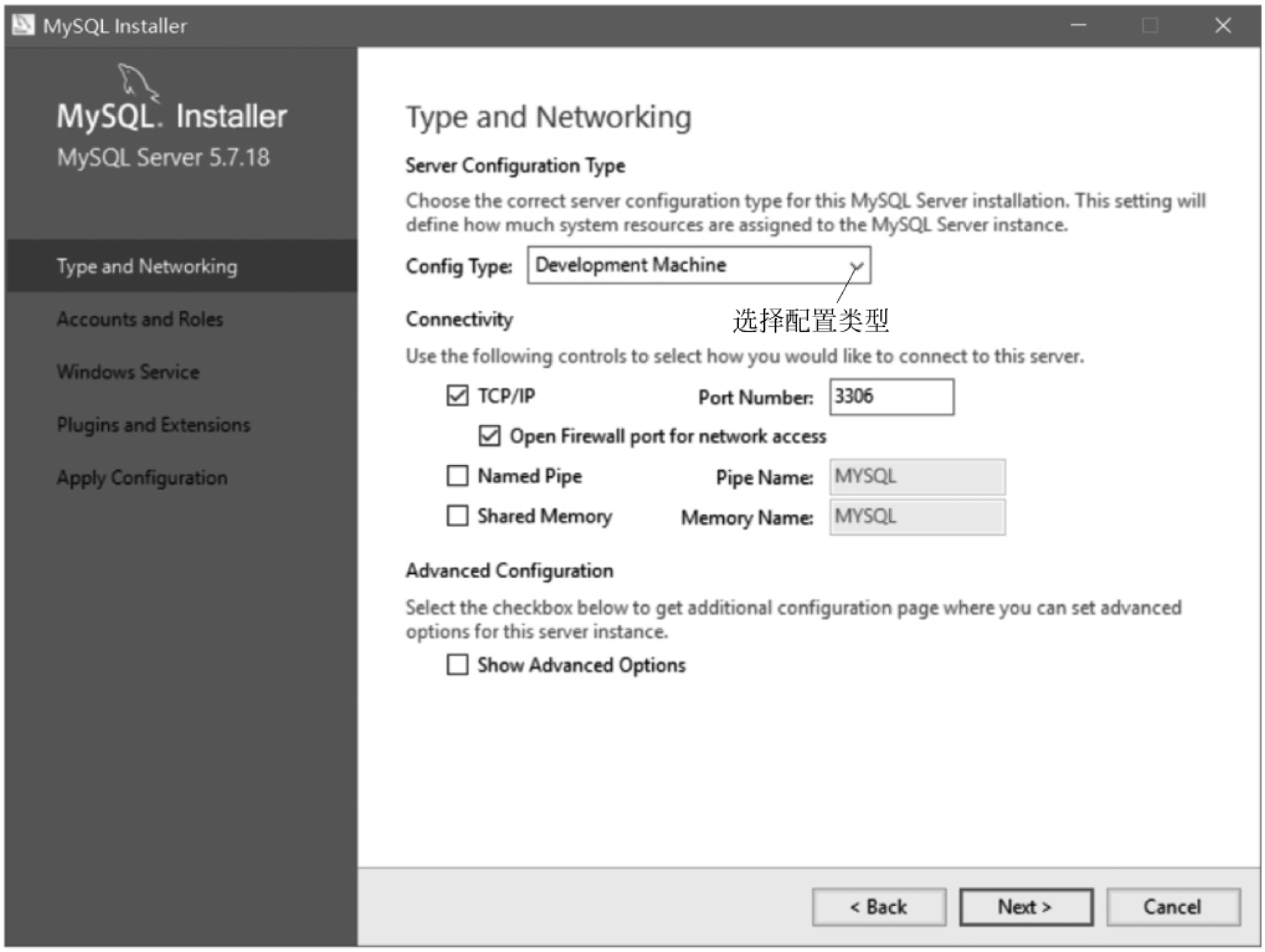


图 A-5　服务器配置类型对话框

- Development Machine(开发机器)：该选项代表典型个人用桌面工作站，假定机器上运行着多个桌面应用程序，将 MySQL 服务器配置成使用最少的系统资源。
- Server Machine(服务器)：该选项代表服务器，MySQL 服务器可以同其他应用程序一起运行，如 FTP、Email 和 Web 服务器。MySQL 服务器配置成使用适当比例的系统资源。
- Dedicated Machine(专用 MySQL 服务器)：该选项代表只运行 MySQL 服务的服务器。假定没有运行其他应用程序。MySQL 服务器配置成使用所有可用的系统资源。

根据用户自己的需要选择配置类型，其他的配置项目保持默认值，单击 Next 按钮进入如图 A-6 所示的账号和用户角色设置对话框。

在图 A-6 所示的对话框中可以进行 root 密码设置，以及添加其他账号等操作。root 密码必须是 4 位以上，根据需要设置 root 密码。此外，还可以单击 Add User 按钮添加其他的账号。

在图 A-6 所示对话框设置完成后，单击 Next 按钮进入图 A-7 所示的配置 Windows 服务对话框。在这里可以将 MySQL 数据库配置成为一个 Windows 服务。Windows 服务可以在后台随着 Windows 的启动而启动，不需要人为干预。其实默认的服务名是 MySQL57。

在图 A-7 所示界面中配置之后不需要再进行其他配置了，只需要单击 Next 按钮即可完成配置，这里不再赘述。

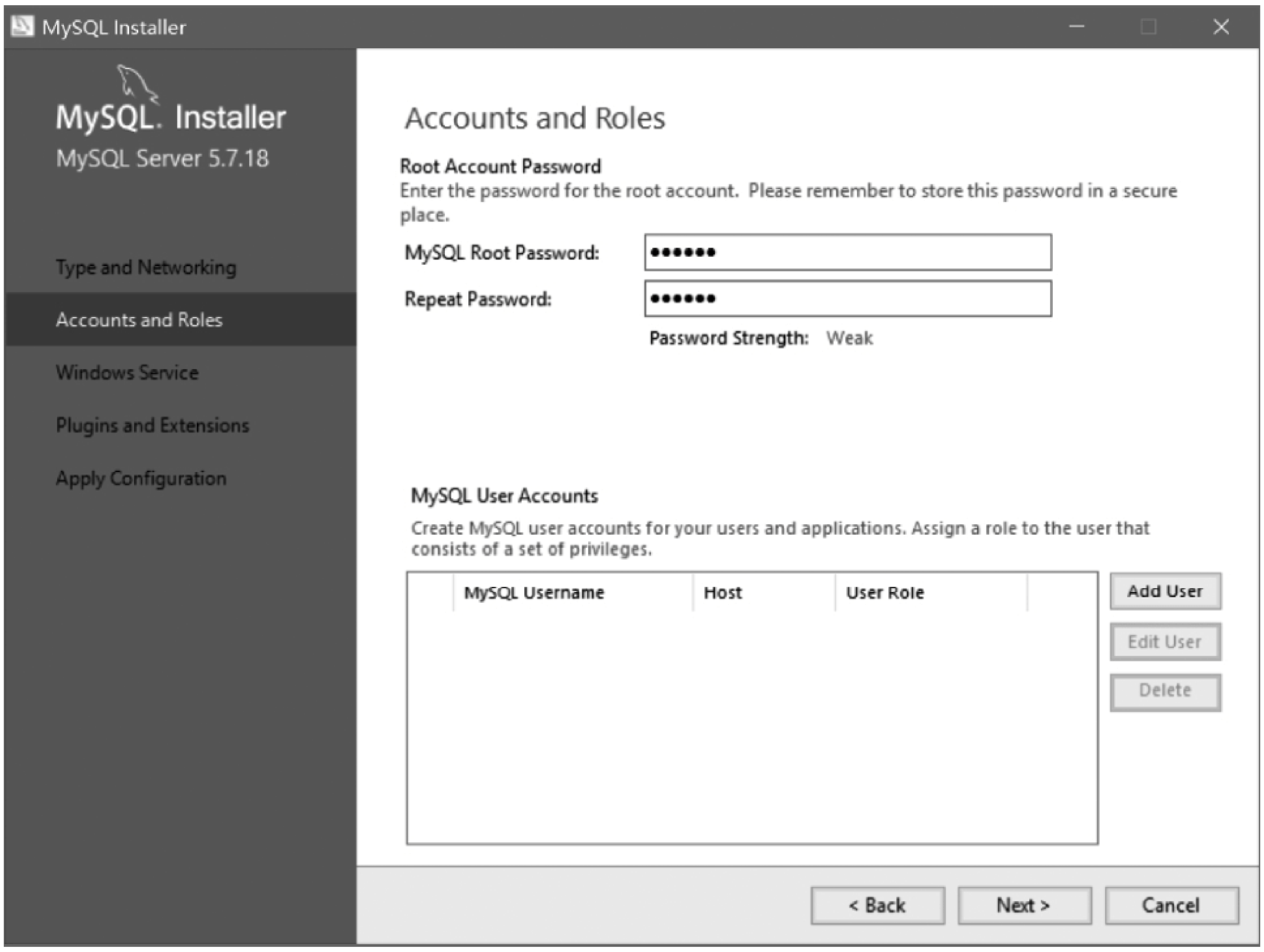


图 A-6 账号和用户角色设置对话框

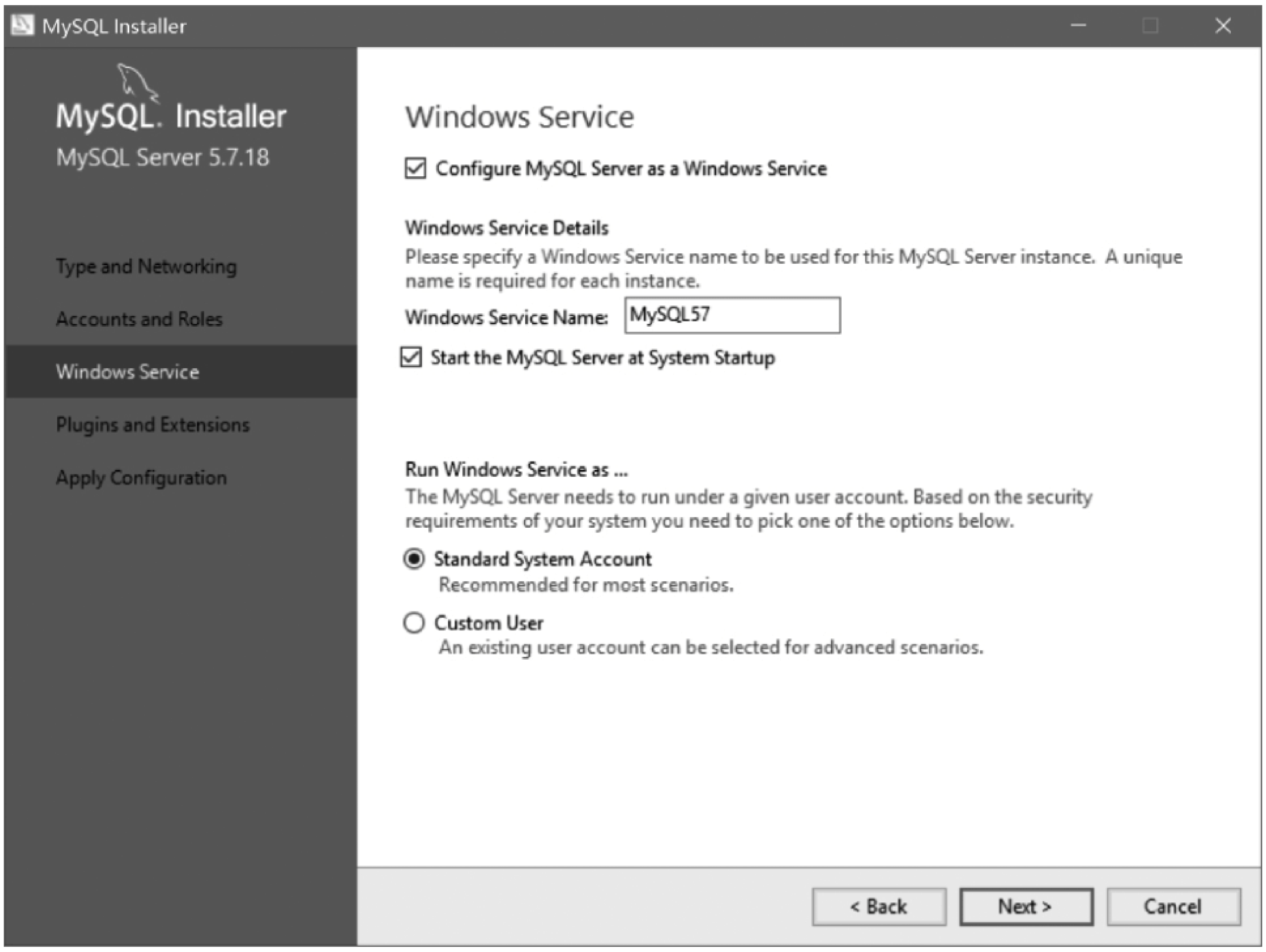


图 A-7 配置 Windows 服务对话框

A.2.2　连接 MySQL 服务器

由于 MySQL 是 C/S(客户端/服务器)结构的,所以应用程序包括它的客户端必须连接到服务器才能使用其服务功能。下面主要介绍 MySQL 本身的客户端如何连接到服务器。

1. 快速连接服务器方式

MySQL for Windows 版本提供一个菜单项目可以快速连接服务器,打开过程为:右击屏幕左下角的 Windows 图标,在“最近添加”中找到 MySQL 5.7 Command Line Client,则会打开一个终端窗口,如图 A-8 所示。

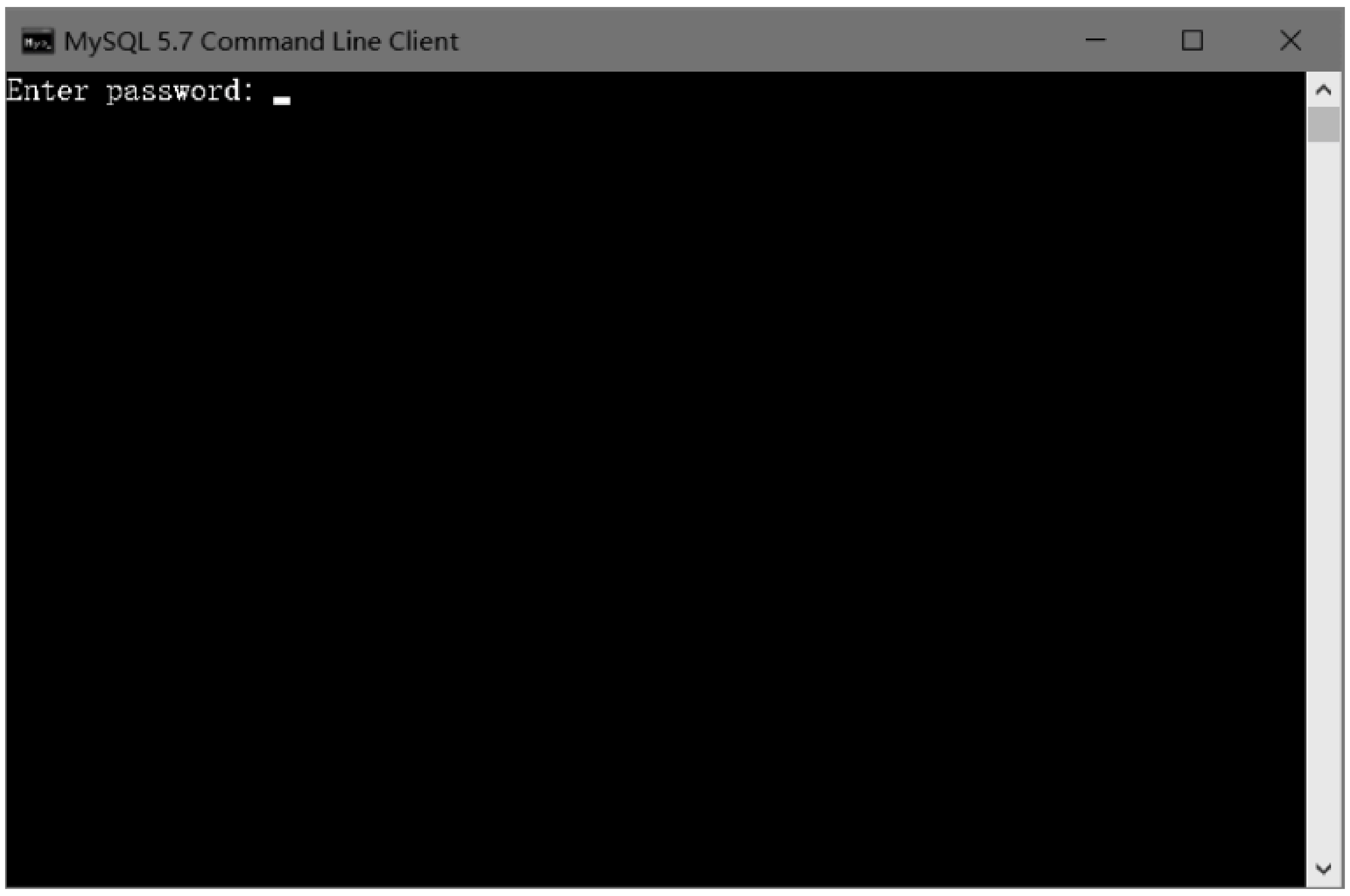


图 A-8　MySQL 命令行客户端

这个工具就是 MySQL 命令行客户端工具,可以使用 MySQL 命令行客户端工具连接到 MySQL 服务器,要求输入 root 密码。输入 root 密码后按 Enter 键,如果密码正确,则连接到 MySQL 服务器,如图 A-9 所示。

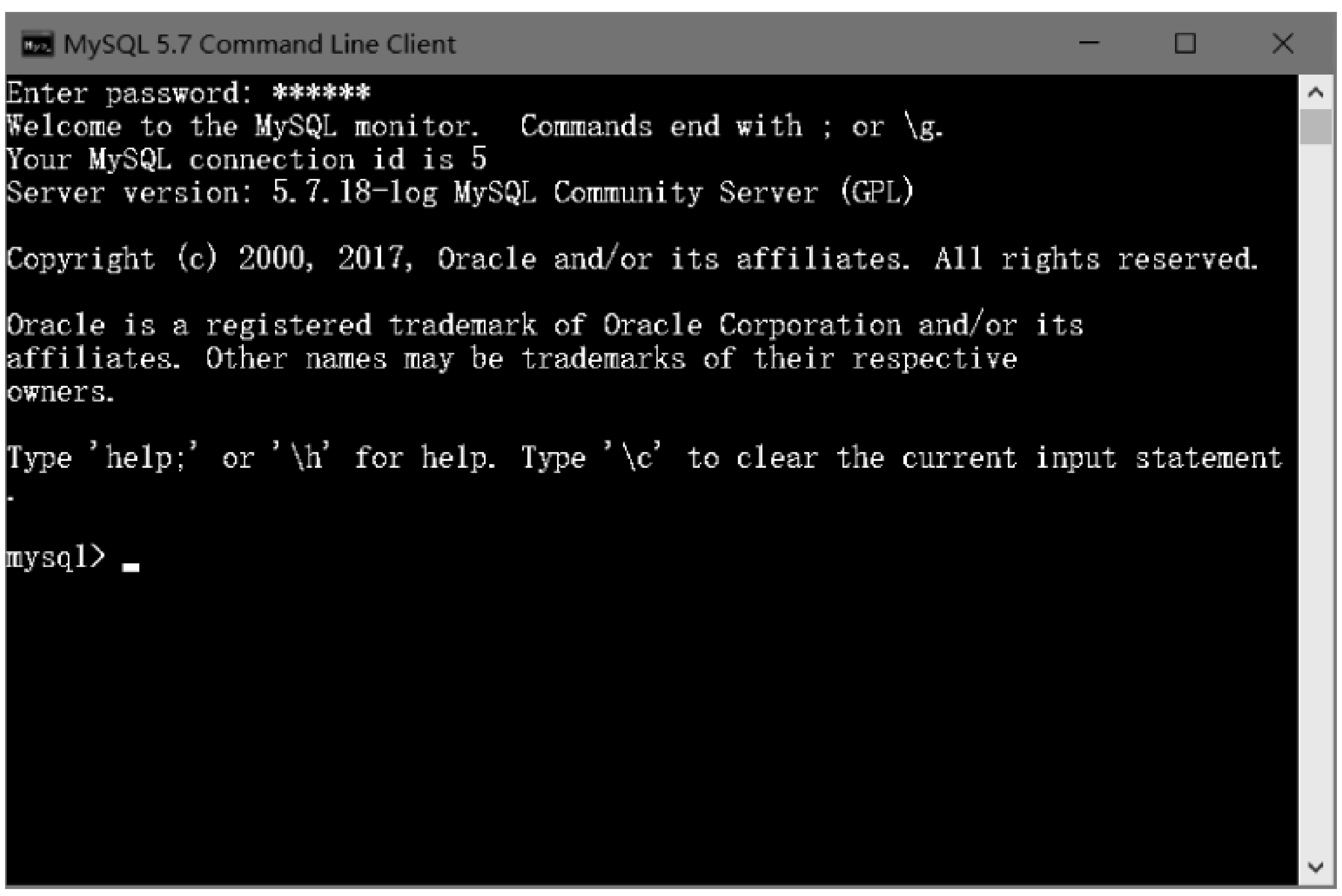


图 A-9　使用命令行客户端连接到服务器

2. 通用的连接方式

快速连接服务器方式连接的是本地数据库，如果服务器不在本地，而是在一个远程主机上，则需要使用通用的连接方式。

首先在操作系统下打开一个终端窗口，Windows 下是命令行工具，在此输入 mysql -h localhost -u root -p 命令。如图 A-10 所示，如果出现“'mysql' 不是内部或外部命令，也不是可运行的程序或批处理文件。”错误提示，则说明在环境变量的 Path 中没有配置 MySQL 的 Path。参考 2.1.2 节，追加 C:\Program Files\MySQL\MySQL Server 5.7\bin 到环境变量 Path 之后。

如果 Path 环境变量添加成功，重新打开命令行，再次输入 mysql -h localhost -u root -p 命令，然后系统会提示输入 root 密码，输入密码后按 Enter 键，如果密码正确，则成功连接到服务器，会看到如图 A-9 所示的界面。

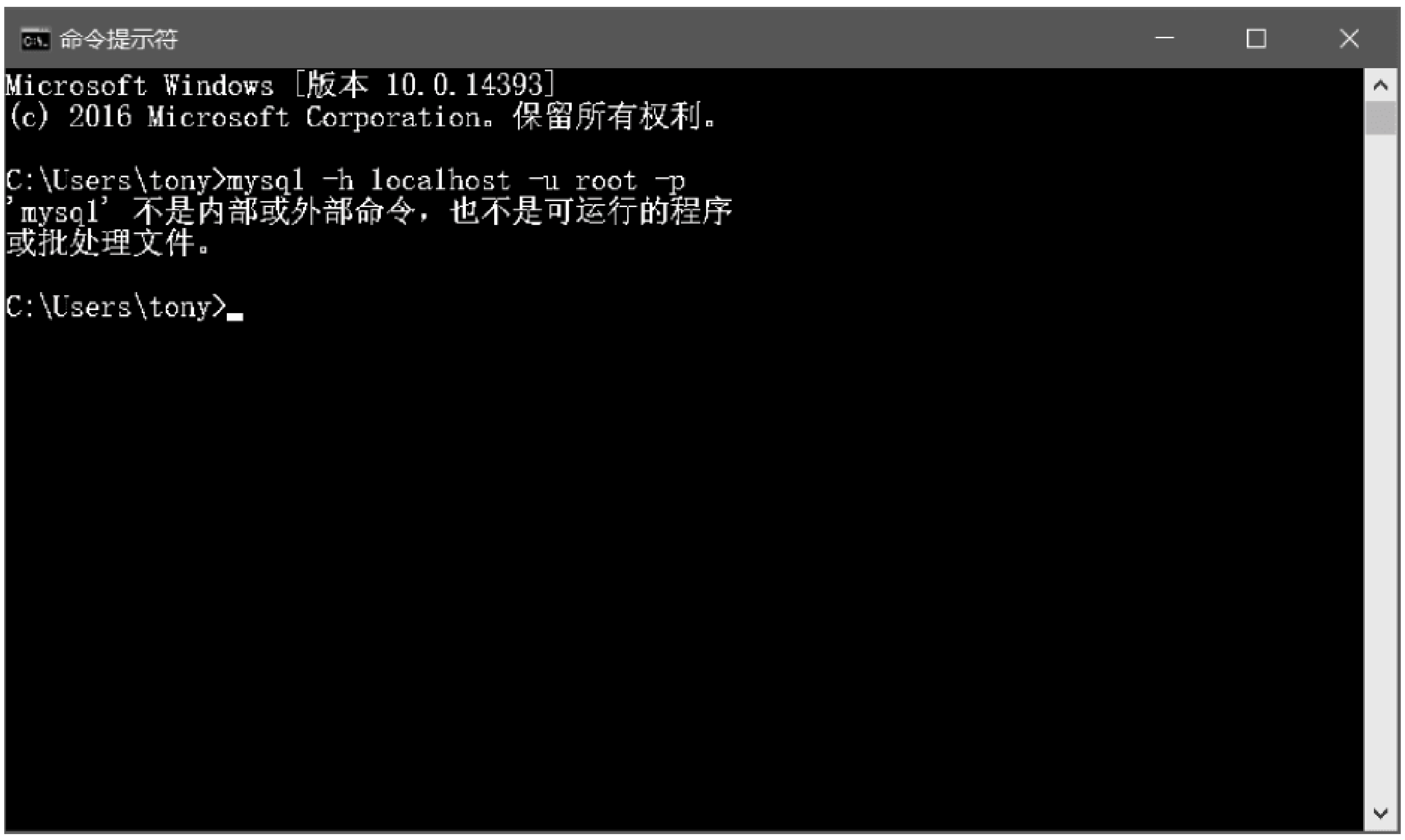


图 A-10　环境变量中没有 MySQL 的 Path

提示　mysql -h localhost -u root-p 命令参数说明：

-h：要连接的服务器主机名或 IP 地址，可以是远程的一个服务器主机，也可以是 -hlocalhost，注意没有空格。

-u：是服务器要验证的用户名，这个用户一定是数据库中存在的，并且具有连接服务器的权限，也可以是-uroot，注意没有空格。

-p：是与上面用户对应的密码，也可以直接输入密码-p123456，123456 是 root 密码。

所以，mysql -h localhost -u root -p 命令也可以替换为 mysql -hlocalhost -uroot -p123456。

A.2.3　常见的管理命令

通过命令行客户端管理 MySQL 数据库，需要了解一些常用的命令。

1. help

第一个应该熟悉的就是 help 命令。使 help 命令能够列出 MySQL 的其他命令的帮助，在命令行客户端中输入 help，不需要分号结尾，直接按下 Enter 键，如图 A-11 所示。这里都是 MySQL 的管理命令，这些命令大部分不需要分号结尾。



图 A-11　使用 help 命令

2. 退出命令

如果要在命令行客户端中退出，可以在命令行客户端中使用 quit 或 exit 命令，如图 A-12 所示。注意这两个命令也不需要分号结尾。

3. 数据库管理

在使用数据库的过程中，有时需要知道服务器中有哪些数据库或自己创建和删除的数据库。查看数据库的命令是“show databases;”，如图 A-13 所示，注意该命令后面是有分号结尾的。

创建数据库可以使用“create database testdb;”命令，如图 A-14 所示，testdb 是自定义数据库名，注意该命令后面是以分号结尾的。

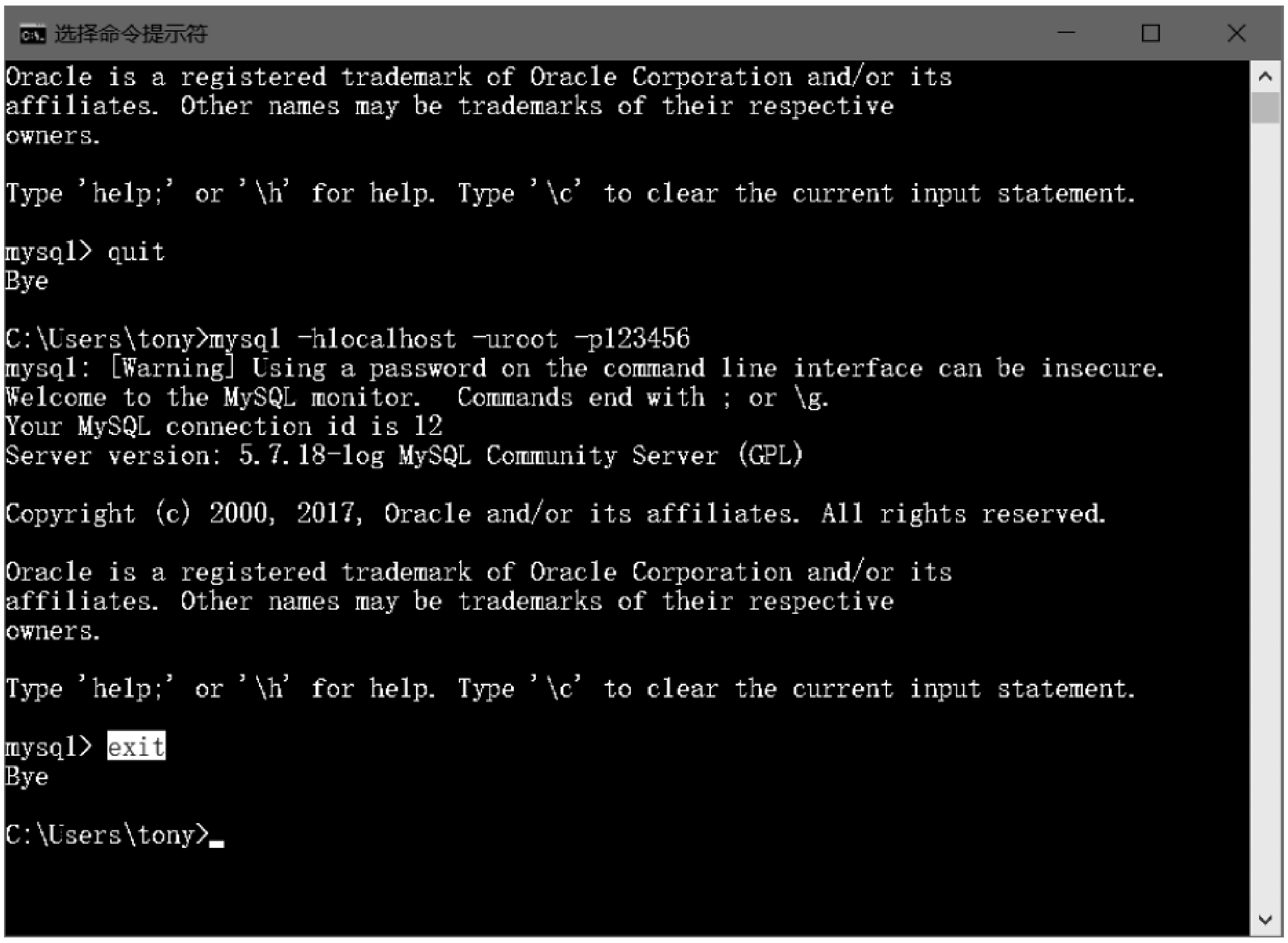


图 A-12　使用退出命令

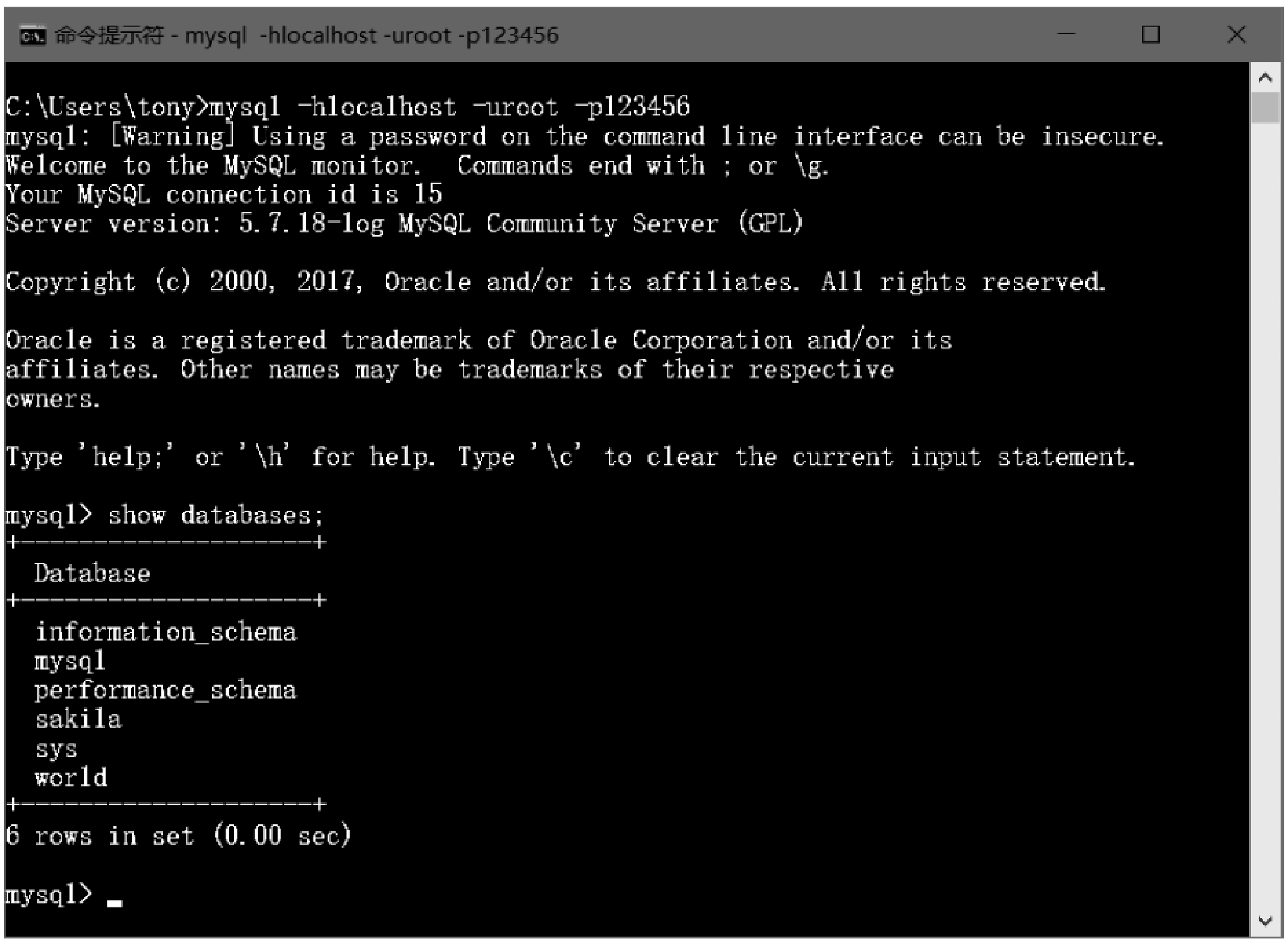


图 A-13　查看数据库信息

想要删除数据库可以使用“drop database testdb;”命令，如图 A-15 所示，testdb 是自定义数据库名，注意该命令后面是以分号结尾的。

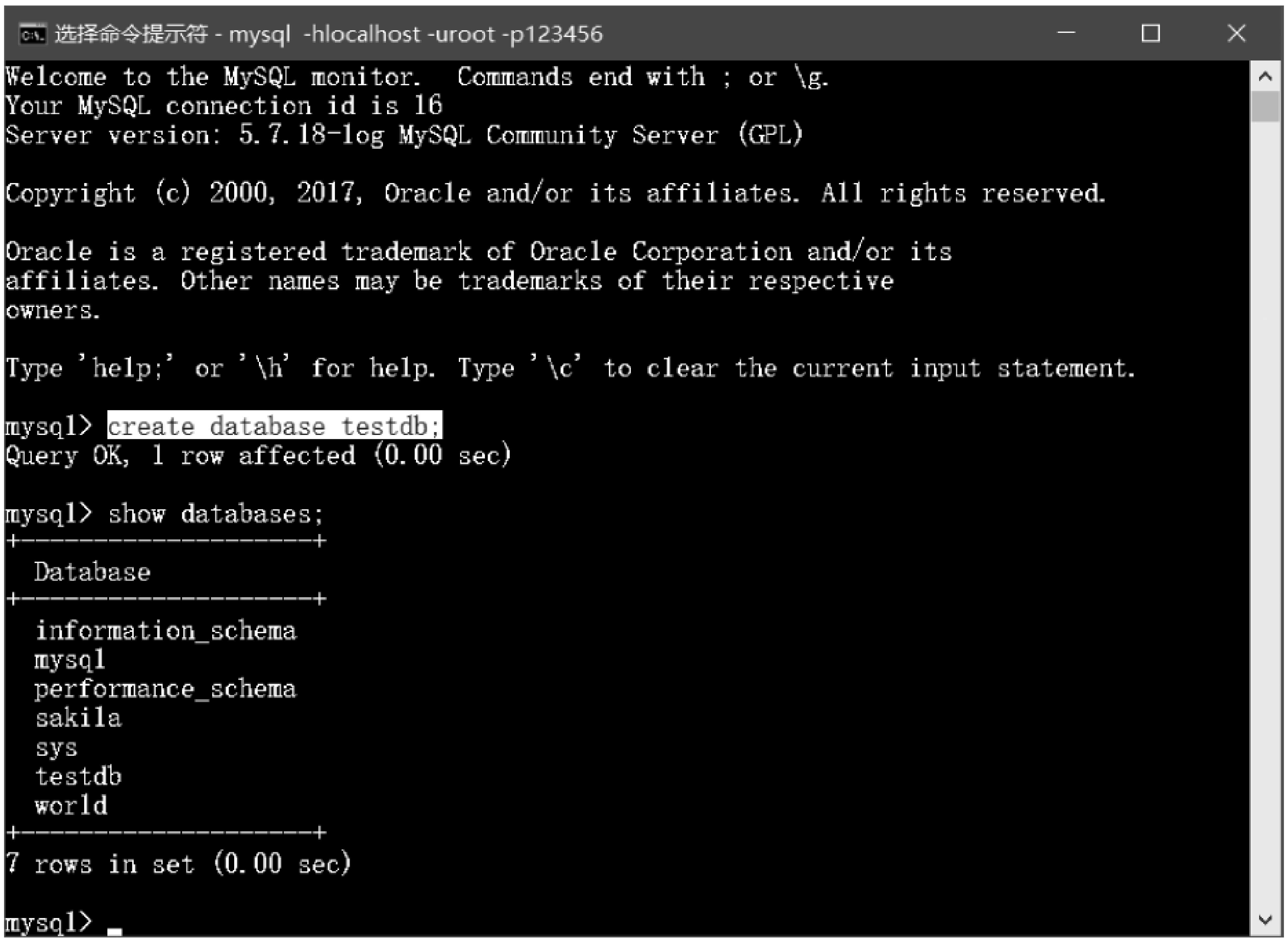


图 A-14　创建数据库

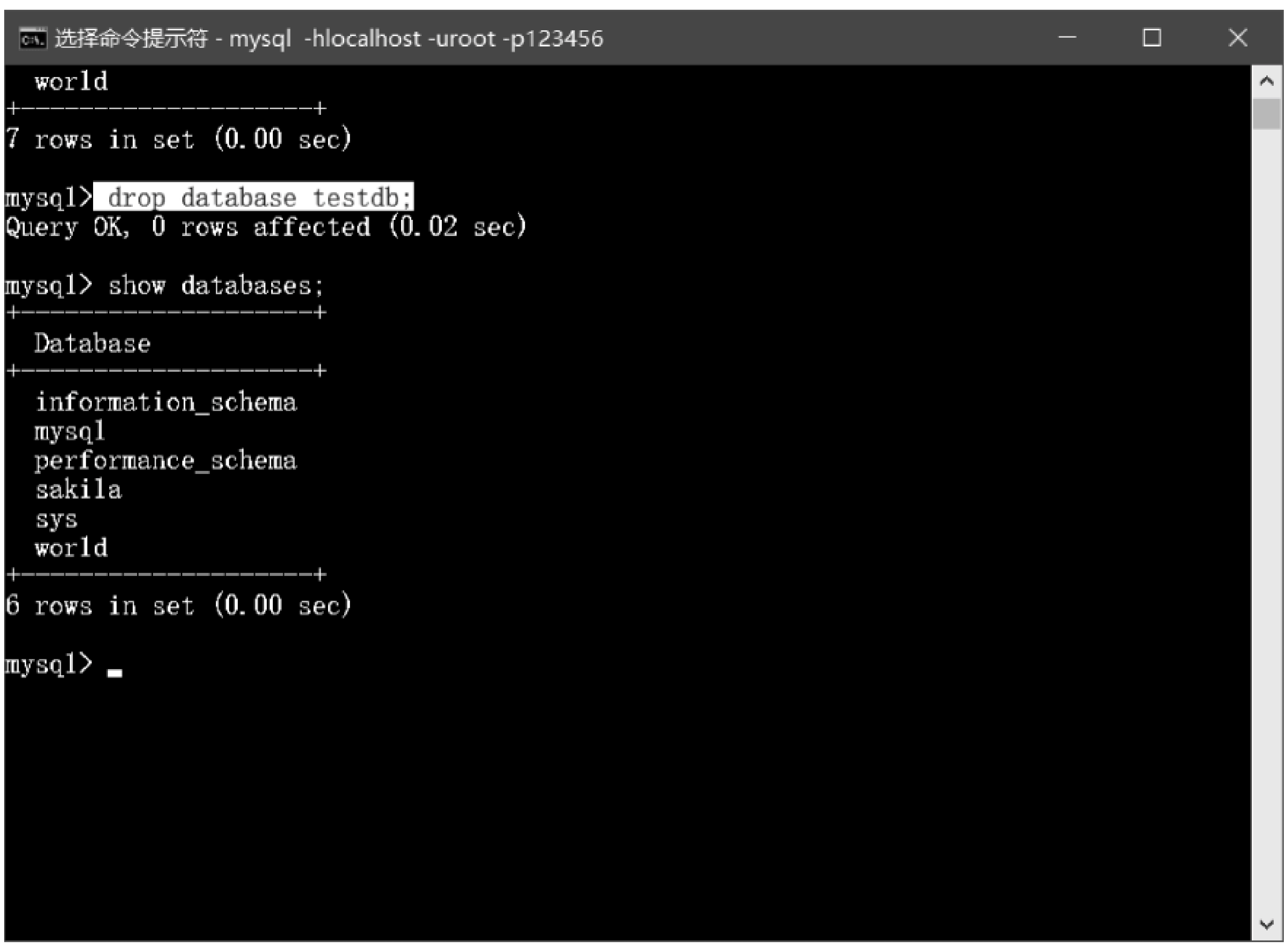


图 A-15　删除数据库

4. 数据表管理

在使用数据库的过程中，有时需要知道某个数据库下有多少个数据表，并想查看表结构

等信息。

查看有多少个数据表的命令是“show tables;”,如图 A-16 所示,注意该命令后面是以分号结尾的。因为一个服务器中有很多数据库,应该先使用 use 命令选择数据库,如图 A-16 所示,use world 命令结尾没有分号。如果没有选择数据库,则会发生错误,如图 A-16 所示。

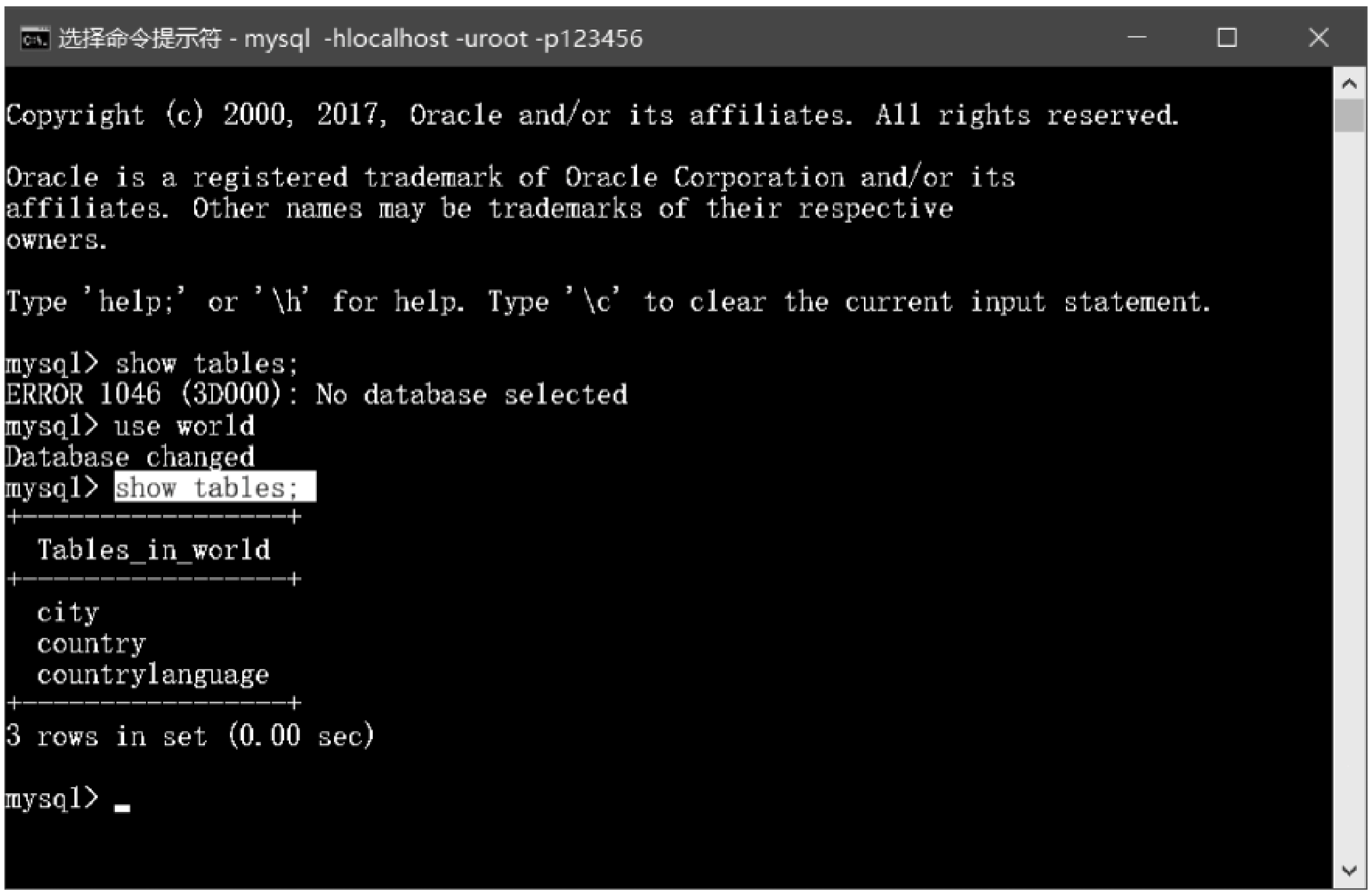


图 A-16 查看数据库中表信息

知道了有哪些表后,还需要知道表结构,可以使用 desc 命令,例如想知道 city 表结构可以使用“desc city;”命令,如图 A-17 所示,注意该命令后面是以分号结尾的。

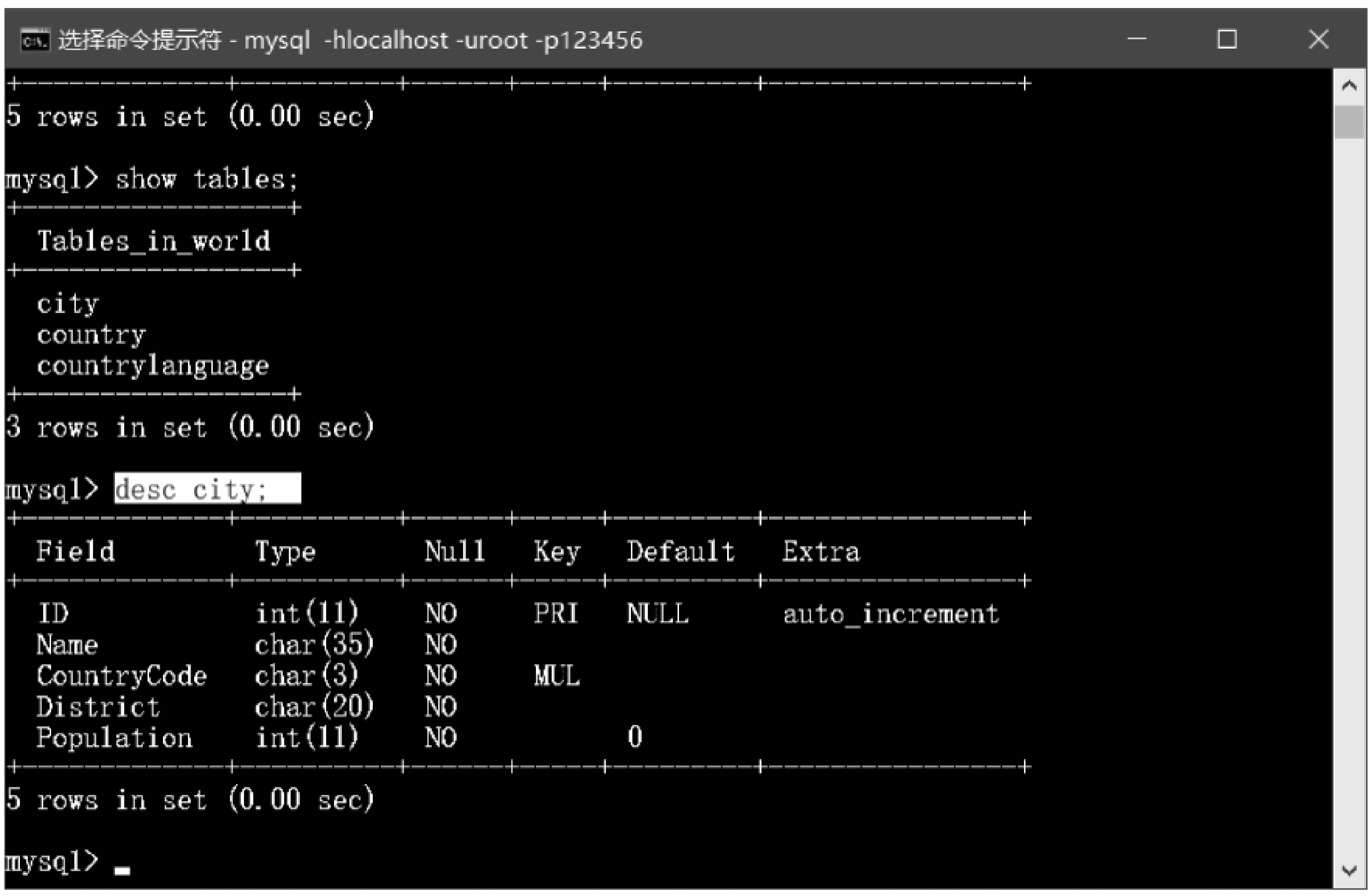


图 A-17 查看表结构

A.3 JDBC 技术

Java 中数据库编程是通过 JDBC 实现的。使用 JDBC 技术涉及 3 种不同的角色：Java 官方、开发人员和数据库厂商，如图 A-18 所示。

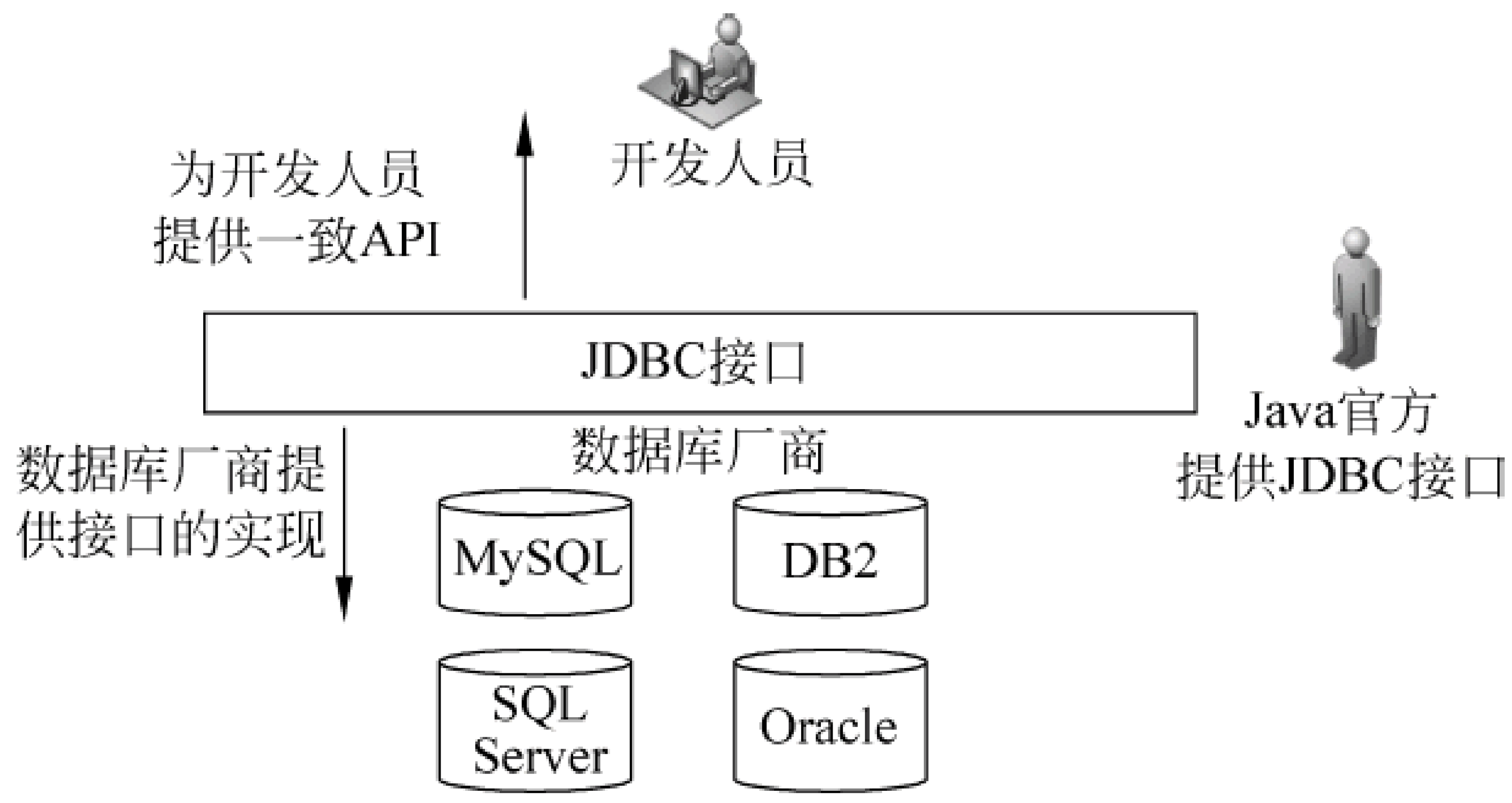


图 A-18 JDBC 技术涉及三种不同的角色

- Java 官方提供 JDBC 接口，如 Connection、Statement 和 ResultSet 等。
- 对于开发人员而言，JDBC 提供了一致的 API，开发人员不用关心实现接口的细节。
- 数据库厂商为了支持 Java 语言使用自己的数据库，他们根据这些接口提供了具体的实现类，这些具体实现类称为 JDBC Driver（JDBC 驱动程序），例如 Connection 是数据库连接接口。如何能够高效地连接数据库或许只有数据库厂商自己清楚，因此他们提供的 JDBC 驱动程序是最高效的，当然针对某种数据库也可能有其他第三方 JDBC 驱动程序。

A.3.1 JDBC API

JDBC API 为 Java 开发者使用数据库提供了统一的编程接口，它由一组 Java 类和接口组成。这种类和接口来自于 java.sql 和 javax.sql 两个包。

- java.sql：该包中的类和接口主要针对基本的数据库编程服务，如创建连接、执行语句、语句预编译和批处理查询等。同时也有一些高级的处理，如批处理更新、事务隔离和可滚动结果集等。
- javax.sql：该包主要为数据库方面的高级操作提供接口和类，提供分布式事务、连接池和行集等。

A.3.2 加载驱动程序

在编程实现数据库连接时，JVM 必须先加载特定厂商提供的数据库驱动程序。使用 Class.forName()方法实现驱动程序加载过程，该方法在前面介绍过。

不同驱动程序的装载方法如下：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");          //JDBC-ODBC 桥接,Java 自带
Class.forName("特定的 JDBC 驱动程序类名");               //数据库厂商提供
```

例如，加载 MySQL 驱动程序代码如下：

```
Class.forName("com.mysql.jdbc.Driver");
```

如果直接这样运行程序，则会抛出如下的 ClassNotFoundException 异常：

```
java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
```

这是因为程序无法找到 MySQL 驱动程序 com.mysql.jdbc.Driver 类，这需要配置当前项目的类路径(Classpath)，在类路径中包含 MySQL 驱动程序。MySQL 驱动程序是在 MySQL 安装目录 Connector.J 5.1 中的 mysql-connector-java-xxx-bin.jar 文件。笔者默认安装驱动程序文件路径如下：

```
"C:\Program Files (x86)\MySQL\Connector.J 5.1\mysql-connector-java-5.1.41-bin.jar"
```

数据库厂商提供的驱动程序一般都是.jar 文件。

提示 一般在发布 java 文件时，会把字节码文件(class 文件)打包成.jar 文件，.jar 文件是一种基于.zip 结构的压缩文件。

在 Eclipse 中将.jar 文件添加到项目的操作步骤：首先在 Eclipse 中选择项目，右击，从弹出的快捷菜单中选择“构建路径”→“配置构建路径”命令，在弹出的对话框中选择“Java 构建路径”→“库”，如图 A-19 所示，其中“添加 JAR”是在当前项目中找.jar 文件，“添加外部 JAR(X)”是在项目之外找.jar 文件。



图 A-19　构建路径对话框

提示 推荐前一种方式(添加 JAR)，它的好处在于当把 Eclipse 项目复制给别人时，这些.jar 文件也一起复制，项目重新编译和运行时，不会发生找不到.jar 文件的情况。而添

加外部 JAR 方法只是添加了一个基于本机的绝对路径，当项目复制给别人时，会发生找不到.jar 文件的情况。

在图 A-19 所示对话框中如果单击“添加 JAR”按钮，则弹出如图 A-20 所示的对话框。通过该对话框只能选择该项目内的.jar 文件，所以要保证所添加的.jar 文件应该在 Eclipse 项目中，如果文件存在，则选择.jar 文件后单击“确定”按钮。

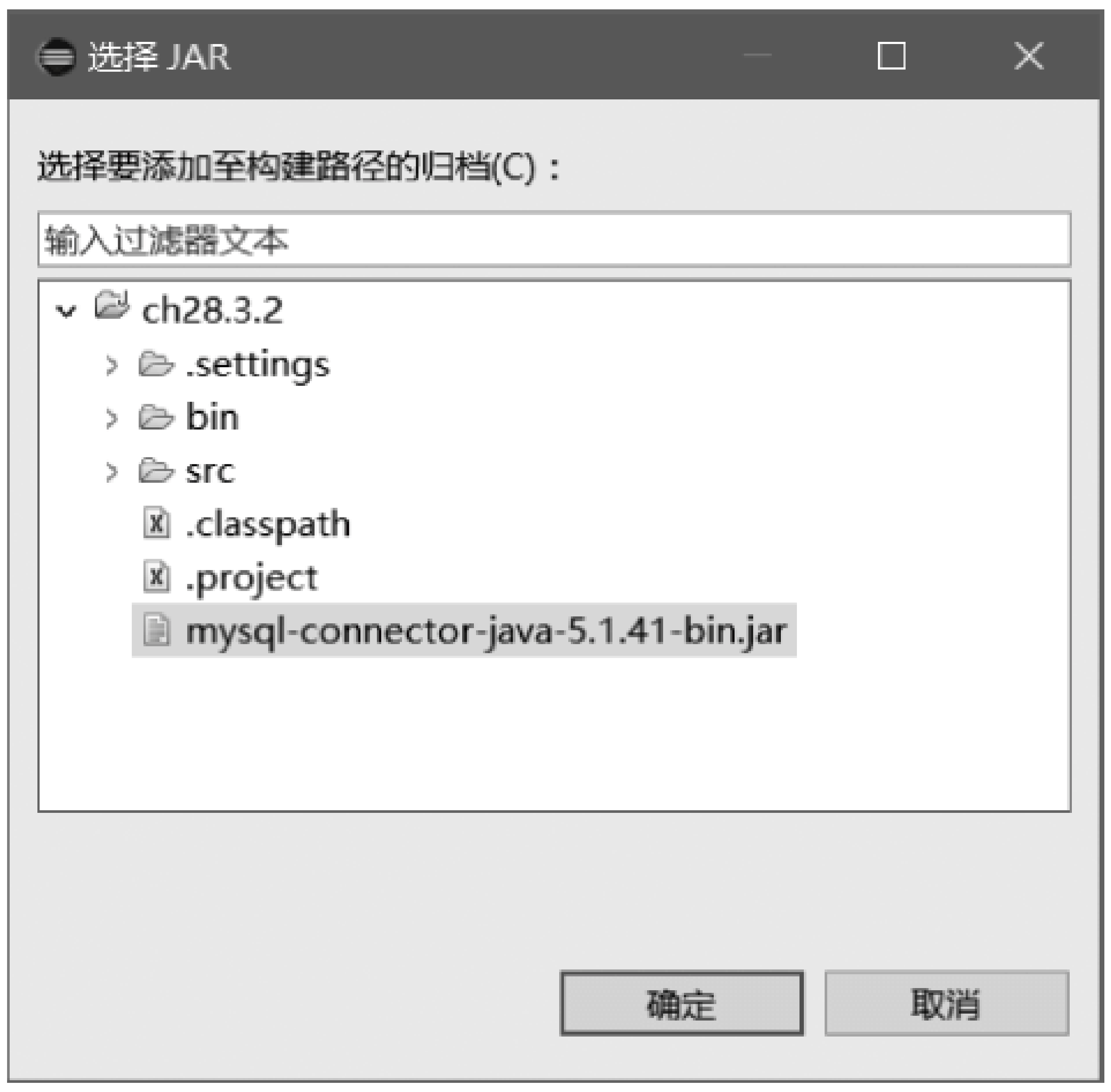


图 A-20　添加.jar 文件到项目类路径

如果文件在 Eclipse 中没有，则需要将其从操作系统的资源管理器复制到 Eclipse 中，如图 A-21 所示。

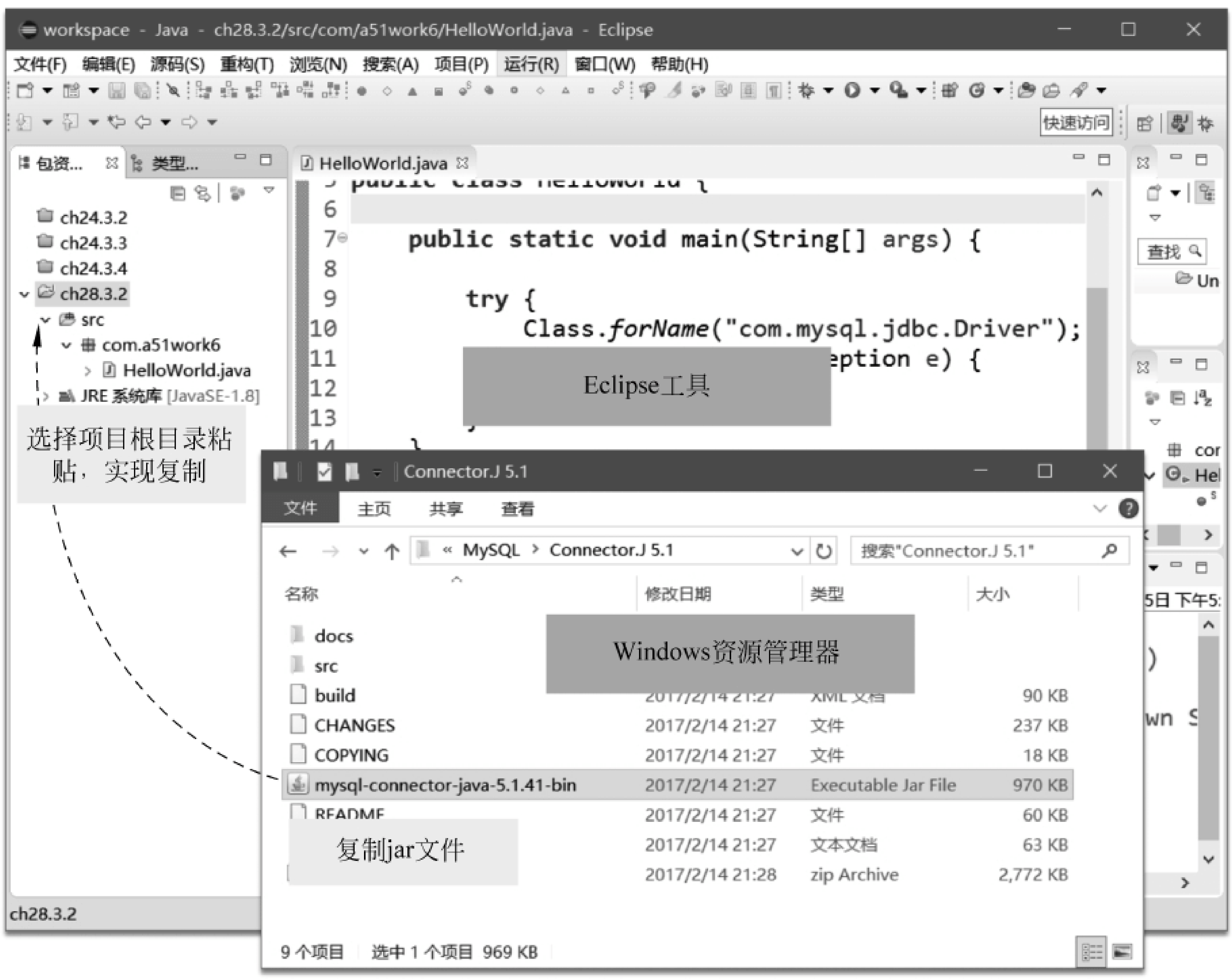


图 A-21　从资源管理器复制文件到 Eclipse

将驱动程序.jar 文件添加到类路径中后，再运行上面的程序看看是否还有 ClassNotFoundException 异常。

A.3.3 建立数据连接

驱动程序加载成功就可以进行数据库连接了。建立数据库连接可以通过调用 DriverManager 类的 getConnection()方法实现。该方法有如下几个重载版本。

- static Connection getConnection(String url)：尝试通过一个 URL 建立数据库连接，调用此方法时，DriverManager 会试图从已注册的驱动中选择恰当的驱动来建立连接。
- static Connection getConnection(String url,Properties info)：尝试通过一个 URL 建立数据库连接，一些连接参数(如 user 和 password)可以按照键值对的形式放置到 info 中，Properties 是 Hashtable 的子类，它是一种 Map 结构。
- static Connection getConnection(String url, String user, String password)：尝试通过一个 URL 建立数据库连接，指定数据库用户名和密码。

上面的几个 getConnection()方法都会抛出受检查的 SQLException 异常，注意处理这个异常。

JDBC 的 URL 类似于其他场合的 URL，它的语法如下：

```
jdbc:<subprotocol>:<subname>
```

这里有 3 部分，它们用冒号隔离。

(1) 协议。jdbc 表示协议，它是唯一的，JDBC 只有这一种协议。

(2) 子协议。主要用于识别数据库驱动程序，也就是说，不同的数据库驱动程序的子协议不同。

(3) 子名。它属于专门的驱动程序，不同驱动程序有不同的子名。

对于不同的数据库，厂商提供的驱动程序和连接的 URL 都不同，如表 A-1 所示。

表 A-1 数据库厂商提供的驱动程序和连接的 URL

数据库名	驱动程序	URL
MS SQL Server	com.microsoft.jdbc.sqlserver.SQLServerDriver	jdbc:microsoft:sqlserver://[ip]:[port];user=[user];password=[password]
JDBC-ODBC	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:[odbcsource]
Oracle thin Driver	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@[ip]:[port]:[sid]
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://ip/database

建立数据连接示例代码如下：

```
//HelloWorld.java 文件
package com.a51work6;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public class HelloWorld {

        public static void main(String[] args) {

            try {
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("驱动程序加载成功…");

            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败…");
                //退出
                return;
            }

            String url = "jdbc:mysql://localhost:3306/MyDB";
            String user = "root";
            String password = "123456";

            try (Connection conn = DriverManager.getConnection(url, user, password)) {①

                System.out.println("数据库连接成功: " + conn);

            } catch (SQLException e) {
                e.printStackTrace();
            }

        }
}
```

上述代码第①行使用 DriverManager 的 getConnection(String url, String user, String password)方法建立数据连接，在 url 中 3306 是数据库端口号，MyDB 是 MySQL 服务器中的数据库。

另外，Connection 对象是通过自动资源管理技术释放资源的。

注意 Connection 对象代表的数据连接不能被 JVM 的垃圾收集器回收，在使用完连接后必须关闭(调用 close()方法)，否则连接会保持一段比较长的时间，直到超时。Java 7 之前都在 finally 模块中关闭数据库连接。Java 7 之后 Connection 接口继承了 AutoCloseable 接口，可以通过自动资源管理技术释放资源。

事实上，上面的代码虽然可以成功建立连接，但是控制台会有警告。警告如下：

```
WARN: Establishing SSL connection without server's identity verification is not recommended.
According to MySQL 5.5.45 , 5.6.26 and 5.7.6 requirements SSL connection must be established by
default if explicit option isn't set. For compliance with existing applications not using SSL
the verify ServerCertificate property is set to 'false'. You need either to explicitly disable
SSL by setting useSSL = false, or set useSSL = true and provide truststore for server certificate
verification.
```

这是由于现在的网络通信为了提高网络完全，都要求使用 SSL[①](Secure Sockets Layer，安全套接层)安全协议。但是由于各种原因，很多服务器并未使用 SSL 安全协议，特别是在学习和测试阶段可以不使用 SSL 安全协议。为此，需要修改 url 连接字符串：

```
"jdbc:mysql://localhost:3306/MyDB?verifyServerCertificate = false&useSSL = false"
```

其中，verifyServerCertificate 设置为 false 表示不进行安全认证，useSSL 设置为 false 表示不使用 SSL 进行网络通信。

数据连接的 url 字符串可以有很多参数对，包括数据库用户名和密码也都可以参数对形式放到 url 字符串中，有的 url 字符串会很长，维护起来不方便，可以把这些参数对放置到 Properties 对象中。示例代码如下：

```
//HelloWorldWithProp.java 文件
package com.a51work6;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class HelloWorldWithProp {

        public static void main(String[] args) {

            try {
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("驱动程序加载成功…");

            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败…");
                //退出
                return;
            }

            String url  =  "jdbc:mysql://localhost:3306/MyDB";
            //创建 Properties 对象
            Properties info  =  new Properties();                            ①
            info.setProperty("user", "root");                               ②
            info.setProperty("password", "123456");                         ③
            info.setProperty("verifyServerCertificate", "false");           ④
            info.setProperty("useSSL", "false");                            ⑤

            try (Connection conn  =  DriverManager.getConnection(url, info)) {   ⑥

                System.out.println("数据库连接成功：" + conn);
```

① SSL 是为网络通信提供安全及数据完整性的是安全协议，SSL 在传输层对网络连接进行加密。

```
                } catch (SQLException e) {
                    e.printStackTrace();
                }

            }
    }
```

上述代码第①行创建 Properties 对象，代码第②～⑤行设置参数，setProperty()方法键和值都是字符串类型。代码第⑥行通过 DriverManager 的 getConnection(String url, Properties info)方法建立数据连接。

但是上述代码还是有不尽如人意的地方，就是这些参数都是“硬编码①”在程序代码中的，程序编译之后不能修改。但是数据库用户名、密码、服务器主机名、端口等，在开发阶段和部署阶段可能完全不同，这些参数信息应该是可以配置的，可以放到一个属性文件中，借助于输入流，可以在运行时读取属性文件内容到 Properties 对象中。具体示例代码如下：

```
//HelloWorldWithPropFile.java 文件
package com.a51work6;

…

public class HelloWorldWithPropFile {

        public static void main(String[] args) {

            //加载驱动程序
            …

            Properties info = new Properties();                        ①
            try {
                InputStream input = HelloWorldWithPropFile.class.getClassLoader()
                        .getResourceAsStream("config.properties");      ②

                info.load(input);                                       ③

            } catch (IOException e) {
                //退出
                return;
            }

            String url = "jdbc:mysql://localhost:3306/MyDB";

            try (Connection conn = DriverManager.getConnection(url, info)) {

                System.out.println("数据库连接成功: " + conn);
```

---

① 硬编码俗称“写死”，指将可变变量用一个固定值来代替的方法，用这种方法编译后，如果以后需要更改此变量就非常困难。

```
                } catch (SQLException e) {
                    e.printStackTrace();
                }

            }
    }
```

上述代码第①行创建一个 Properties 对象。代码第②行获得 config.properties 属性文件输入流对象,属性文件一般在 src 目录,与源代码文件放置在一起,但是编译时,这些文件会被复制到字节码文件所在的目录中,这种目录称为资源目录,获得资源目录要通过 Java 反射机制,HelloWorldWithPropFile.class.getClassLoader().getResourceAsStream("config.properties")语句能够获得运行时 config.properties 的文件输入流对象。

代码第③行是从流中加载信息到 Properties 对象中。

config.properties 文件内容如下:

```
#Tony
user = root
password = 123456
useSSL = false
verifyServerCertificate = false
```

在开发和部署阶段使用文本编辑器修改该文件,不需要修改程序代码。

A.3.4 3 个重要接口

下面重点介绍 JDBC API 中最重要的 3 个接口:Connection、Statement 和 ResultSet。

1. Connection 接口

java.sql.Connection 接口的实现对象代表与数据库的连接,也就是在 Java 程序和数据库之间建立连接。Connection 接口中常用的方法如下。

- Statement createStatement():创建一个语句对象,语句对象用来将 SQL 语句发送到数据库。
- PreparedStatement prepareStatement(String sql):创建一个预编译的语句对象,用来将参数化的 SQL 语句发送到数据库,参数包含一个或者多个问号“?”占位符。
- CallableStatement prepareCall(String sql):创建一个调用存储过程的语句对象,参数是调用的存储过程,参数包含一个或者多个问号“?”占位符。
- close():关闭到数据库的连接,在使用完连接后必须关闭,否则连接会保持一段比较长的时间,直到超时。
- isClosed():判断连接是否已经关闭。

2. Statement 接口

java.sql.Statement 称为语句对象,它提供用于向数据库发出的 SQL 语句,并且给出访问结果。Connection 接口提供了生成 Statement 的方法,一般情况下通过 connection.createStatement()方法就可以得到 Statement 对象。

有 3 个 Statement 接口：java.sql.Statement、java.sql.PreparedStatement 和 java.sql.Callable Statement。其中，PreparedStatement 继承 Statement 接口，CallableStatement 继承 Prepared Statement 接口。Statement 实现对象用于执行基本的 SQL 语句，PreparedStatement 实现对象用于执行预编译的 SQL 语句，CallableStatement 实现对象用于调用数据库中的存储过程。

注意 预编译 SQL 语句是在程序编译时一起进行编译，这样的语句在数据库中执行时不需要编译过程，直接执行 SQL 语句，所以速度很快。在预编译 SQL 语句时会有一些程序执行时才能确定的参数，这些参数采用“?”占位符，直到运行时再用实际参数替换。

Statement 提供了许多方法，最常用的方法如下。

- executeQuery()：运行查询语句，返回 ResultSet 对象。
- executeUpdate()：运行更新操作，返回更新的行数。
- close()：关闭语句对象。
- isClosed()：判断语句对象是否已经关闭。

Statement 对象用于执行不带参数的简单 SQL 语句，它的典型使用如下：

```
Connection conn = DriverManager.getConnection("jdbc:odbc:accessdb", "admin", "admin");
Statement stmt = conn.createStatement();
ResultSet rst = stmt.executeQuery("select userid, name from user");
```

PreparedStatement 对象用于执行带参数的预编译 SQL 语句，它的典型使用如下：

```
Connection conn = DriverManager.getConnection("jdbc:odbc:accessdb", "admin", "admin");
PreparedStatement pstmt = conn.prepareStatement("insert into user values (?,?)");
pstmt.setInt(1,10);                        //绑定第一个参数
pstmt.setString(2,"guan");                 //绑定第二个参数
pstmt.executeUpdate();                     //执行 SQL 语句
```

上述 SQL 语句"insert into user values(?,?)"在 Java 源程序编译时一起编译，两个问号占位符所代表的参数在运行时绑定。

注意 绑定参数时需要注意两个问题：绑定参数顺序和绑定参数的类型，绑定参数索引是从 1 开始的，而不是从 0 开始的。根据绑定参数的类型不同选择对应的 set 方法。

CallableStatement 对象用于执行对数据库已存储过程的调用，它的典型使用如下：

```
Connection conn = DriverManager.getConnection("jdbc:odbc:accessdb", "admin", "admin");
strSQL = "{call proc_userinfo(?,?)}";
java.sql.CallableStatement sqlStmt = conn.prepareCall(strSQL);
sqlStmt.setString(1,"tony");
sqlStmt.setString(2,"tom");
//执行存储过程
int i = sqlStmt.executeUpdate();
```

3. ResultSet 接口

在 Statement 执行 SQL 语句时，如果是 select 语句，则会返回结果集，结果集通过接口 java.sql.ResultSet 描述，它提供了逐行访问结果集的方法，通过该方法能够访问结果集中不同字段的内容。

ResultSet 提供了检索不同类型字段的方法，最常用的方法介绍如下。

- close()：关闭结果集对象。
- isClosed()：判断结果集对象是否已经关闭。
- next()：将结果集的光标从当前位置向后移一行。
- getString()：获得在数据库里是 CHAR 或 VARCHAR 等字符串类型的数据，返回值类型是 String。
- getFloat()：获得在数据库里是浮点类型的数据，返回值类型是 float。
- getDouble()：获得在数据库里是浮点类型的数据，返回值类型是 double。
- getDate()：获得在数据库里是日期类型的数据，返回值类型是 java. sql. Date。
- getBoolean()：获得在数据库里是布尔数据的类型，返回值类型是 boolean。
- getBlob()：获得在数据库里是 Blob(二进制大型对象)类型的数据，返回值类型是 Blob 类型。
- getClob()：获得在数据库里是 Clob(字符串大型对象)类型的数据，返回值类型是 Clob。

这些方法要求有列名或者列索引，如 getString()方法的两种情况：

```
public String getString(int columnIndex) throws SQLException
public String getString(String columnName) throws SQLException
```

方法 getXXX 提供了获取当前行中某列值的途径，在每一行内，可按任何次序获取列值。使用列索引有时会比较麻烦，这个顺序是 select 语句中的顺序：

```
select * from user
select userid, name from user
select name,userid from user
```

注意 columnIndex 列索引是从 1 开始的，而不是从 0 开始的。这个顺序与 select 语句有关，如果 select 使用 * 返回所有字段，如 select * from user 语句，那么列索引是数据表中字段的顺序；如果 select 指定具体字段，如 select userid, name from user 或 select name,userid from user，那么列索引是 select 指定字段的顺序。

ResultSet 示例代码如下：

```
//HelloWorldWithPropFile.java 文件
…
String url = "jdbc:mysql://localhost:3306/MyDB";

try (        //自动资源管理技术释放资源
            Connection conn = DriverManager.getConnection(url, info);
            Statement stmt = conn.createStatement();
            ResultSet rst = stmt.executeQuery("select name,userid from user")) {

        while (rst.next()) {
            System.out.printf("name: %s  id: %d\n", rst.getString("name"), rst.getInt (2));
        }
```

```
    } catch (SQLException e) {
            e.printStackTrace();
    }
```

从上述代码可见,Connection 对象、Statement 对象和 ResultSet 对象的释放采用自动资源管理。

在遍历结果集时使用了 rst.next()方法,next()是将结果集光标从当前位置向后移一行,结果集光标最初位于第一行之前;第一次调用 next 方法使第一行成为当前行;第二次调用使第二行成为当前行,以此类推。如果新的当前行有效,则返回 true;如果不存在下一行,则返回 false。

注意 Connection 对象、Statement 对象和 ResultSet 对象都不能被 JVM 的垃圾收集器回收,在使用完后都必须关闭(调用它们的 close()方法)。Java 7 之前都在 finally 模块中关闭释放资源。Java 7 之后它们都继承了 AutoCloseable 接口,可以通过自动资源管理技术释放资源。

A.4 案例:数据 CRUD 操作

对数据库表中数据可以进行 4 类操作:数据插入(Create)、数据查询(Read)、数据更新(Update)和数据删除(Delete),也是俗称的"增、删、改、查",也称为 CRUD 操作。

本节通过一个案例介绍如何通过 JDBC 技术实现 Java 对数据的 CRUD 操作。

A.4.1 数据库编程的一般过程

在讲解案例之前,有必要先介绍一下通过 JDBC 进行数据库编程的一般过程。

如图 A-22 所示是数据库编程的一般过程,其中查询(Read)过程最多需要 7 个步骤,修改(C 插入、U 更新、D 删除)过程最多需要 5 个步骤。这个过程采用了预编译语句对象进行数据操作,所以有可能进行绑定参数,见第 4 步骤。

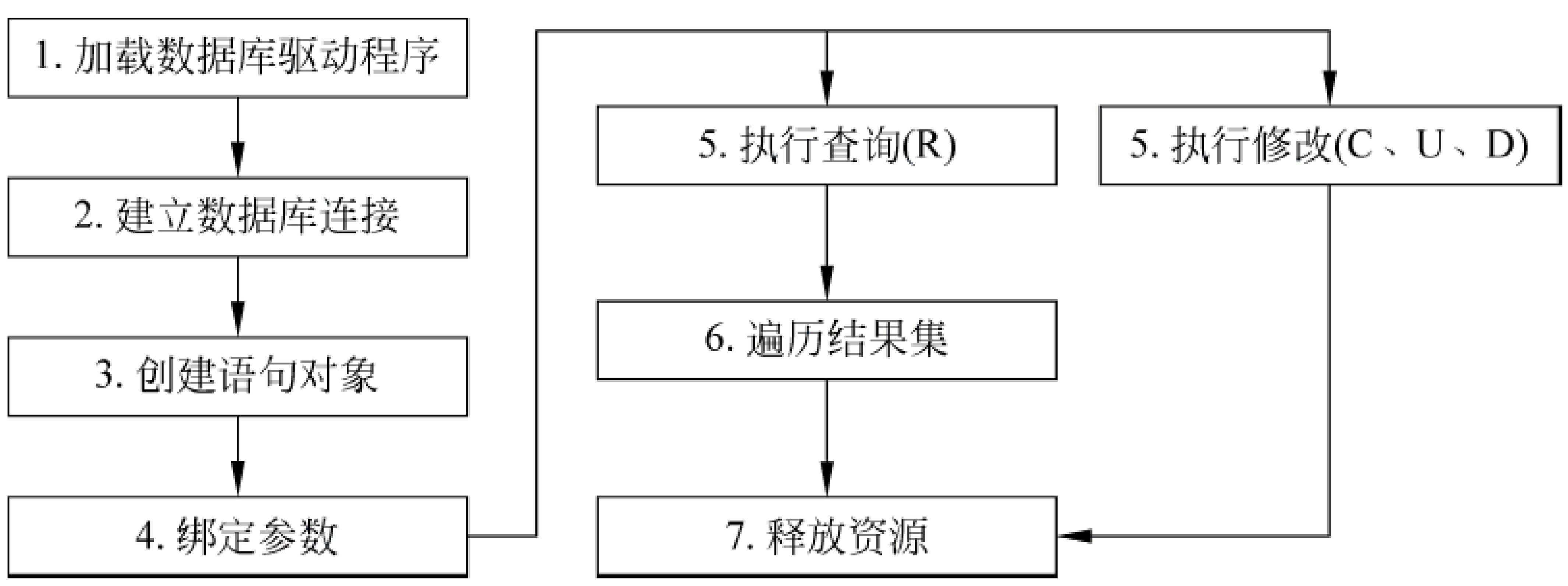


图 A-22 数据库编程的一般过程

上述步骤是基本的一般步骤,实际情况会有所变化,例如没有参数需要绑定,则第 4 步就省略了。另外,如果 Connection 对象、Statement 对象和 ResultSet 对象都采用自动资源管理技术释放资源,那么第 7 步也可以省略。

A.4.2 数据查询操作

为了介绍数据查询操作案例，这里准备了一个 User 表，它有两个字段 name 和 userid，如表 A-2 所示。

表 A-2 User 表结构

字 段 名	类 型	是否可以为 Null	主 键
name	varchar(20)	是	否
userid	int	否	是

下面介绍实现如下两条 SQL 语句查询功能：

```
select name,userid from user where userid > ? order by userid    //有条件查询
select max(userid) from user                                     //使用 max 等函数,无条件查询
```

1. 有条件查询

实现代码如下：

```
//CRUDSample.java 文件
package com.a51work6;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;

public class CRUDSample {

        //连接数据库 url
        static String url;
        //创建 Properties 对象
        static Properties info = new Properties();

        //1. 驱动程序加载
        static {                                                        ①
            //获得属性文件输入流
            InputStream input
                = CRUDSample.class.getClassLoader().getResourceAsStream("config.properties");

            try {
                //加载属性文件内容到 Properties 对象
                info.load(input);
                //从属性文件中取出 url
```

```
                url = info.getProperty("url");
                //Class.forName("com.mysql.jdbc.Driver");
                //从属性文件中取出 driver
                String driverClassName = info.getProperty("driver");
                Class.forName(driverClassName);
                System.out.println("驱动程序加载成功…");
            } catch (ClassNotFoundException e) {
                System.out.println("驱动程序加载失败…");
            } catch (IOException e) {
                System.out.println("加载属性文件失败…");
            }
        }                                                                    ②

        public static void main(String[] args) {

            //查询数据
            read();
        }

        //数据查询操作
        public static void read() {                                          ③

            Connection conn = null;
            PreparedStatement pstmt = null;
            ResultSet rs = null;

            try {
                //2. 创建数据库连接
                conn = DriverManager.getConnection(url, info);
                //3. 创建语句对象
                pstmt = conn.prepareStatement("select name,userid from"
                        + "user where userid > ? order by userid");
                //4. 绑定参数
                pstmt.setInt(1, 0);
                //5. 执行查询(R)
                rs = pstmt.executeQuery();
                //6. 遍历结果集
                while (rs.next()) {
                    System.out.printf("id: %d  name: %s\n", rs.getInt(2),
rs.getString("name"));
                }

            } catch (SQLException e) {
                e.printStackTrace();
            } finally {            // 7. 释放资源                              ④
                if (rs != null) {
                    try {
                        rs.close();
                    } catch (SQLException e) {
                    }
```

```
                    }
                    if (pstmt != null) {
                        try {
                            pstmt.close();
                        } catch (SQLException e) {
                        }
                    }
                    if (conn != null) {
                        try {
                            conn.close();
                        } catch (SQLException e) {
                        }
                    }
                }                                                               ⑤
            }

    }
```

上述代码第①～②行是静态代码块，在静态代码块中读取属性文件内容到 Properties 对象和加载驱动程序，这两个操作只需执行一次，所以它们最好放到静态代码块中。另外，需要注意本例中将驱动程序类名和数据库连接的 url 字符串都放到属性文件 config .properties 中，这样更加方便配置。config. properties 内容如下：

```
driver = com.mysql.jdbc.Driver
url = jdbc:mysql://localhost:3306/MyDB
user = root
password = 123456
useSSL = false
verifyServerCertificate = false
```

上述代码第③行的 read()方法是数据查询方法，查询完成之后采用 finally 代码块释放资源，见代码第④～⑤行。本例也可以使用自动资源管理技术，但会引起 try 语句发生嵌套，反而会有些麻烦。

2. 无条件查询

主要实现代码如下：

```
//查询最大的用户 Id
public static int readMaxUserId() {

        int maxId = 0;
        try (
                //2. 创建数据库连接
                Connection conn = DriverManager.getConnection(url, info);
```

```
                //3. 创建语句对象
                PreparedStatement pstmt = conn.prepareStatement("select max (userid) from
user");
                //4. 绑定参数
                //pstmt.setInt(1, 0);
                //5. 执行查询(R)
                ResultSet rs = pstmt.executeQuery()) {
            //6. 遍历结果集
            if(rs.next()) {
                maxId = rs.getInt(1);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }

        return maxId;
}
```

上述代码使用了自动资源管理技术,由于没有参数需要绑定,所以 ResultSet 对象可以与 Connection 对象和 PreparedStatement 对象放在一个 try 代码块中进行管理。而前面的有条件查询 read()方法则不行。

A.4.3 数据修改操作

数据修改操作包括数据插入、数据更新和数据删除。

1. 数据插入

数据插入代码如下:

```
//数据插入操作
public static void create() {

   try (   //2. 创建数据库连接
            Connection conn = DriverManager.getConnection(url, info);
           //3. 创建语句对象
            PreparedStatement pstmt
               = conn.prepareStatement("insert into user (userid, name) values (?,?)")) {      ①

            //查询最大值
            int maxId = readMaxUserId();

            //4. 绑定参数
            pstmt.setInt(1, ++maxId);                                                           ②
            pstmt.setString(2, "Tony" + maxId);                                                 ③
            //5. 执行修改(C、U、D)
            int affectedRows = pstmt.executeUpdate();                                           ④

            System.out.printf("成功插入 %d 条数据.\n", affectedRows);
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

代码第①行创建插入语句对象，其中有两个占位符。因此需要绑定参数，代码第②行绑定第一个参数，代码第③行绑定第二个参数。代码第④行的 executeUpdate()方法执行 SQL 语句，该方法与查询方法 executeQuery()不同。executeUpdate()方法返回的是整数——成功影响的记录数，即成功插入记录数。

2. 数据更新

数据更新代码如下：

```
//数据更新操作
public static void update() {

        try ( //2. 创建数据库连接
                Connection conn = DriverManager.getConnection(url, info);
                //3. 创建语句对象
                PreparedStatement pstmt
                    = conn.prepareStatement("update user set name = ? where userid > ?")) {

            //4. 绑定参数
            pstmt.setString(1, "Tom");
            pstmt.setInt(2, 30);
            //5. 执行修改(C、U、D)
            int affectedRows = pstmt.executeUpdate();

            System.out.printf("成功更新%d条数据.\n", affectedRows);

        } catch (SQLException e) {
            e.printStackTrace();
        }
}
```

3. 数据删除

数据删除代码如下：

```
//数据删除操作
public static void delete() {

        try ( //2. 创建数据库连接
                Connection conn = DriverManager.getConnection(url, info);
                //3. 创建语句对象
                 PreparedStatement pstmt = conn.prepareStatement("delete from user where
userid = ?")) {

            //查询最大值
            int maxId = readMaxUserId();
```

```
            //4. 绑定参数
            pstmt.setInt(1, maxId);
            //5. 执行修改(C、U、D)
            int affectedRows = pstmt.executeUpdate();

            System.out.printf("成功删除 %d 条数据.\n", affectedRows);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

数据更新、数据删除与数据插入程序结构上非常类似，差别主要在于 SQL 语句与绑定参数的不同，具体代码不再解释。

本章小结

本章首先介绍 MySQL 数据库的安装、配置和日常的管理命令，然后重点讲解了 JDBC 数据库编程技术。读者需要重点掌握 3 个主要接口，熟悉一般数据库编程过程。

附录 B

APPENDIX B

项目实战：开发 PetStore 宠物商店项目

前面学习的 Java 知识只有通过项目贯穿起来，才能将书本中的知识变成自己的。通过项目实战读者能够了解软件开发流程，了解所学知识在实际项目中的使用情况，哪些是重点的，哪些是了解的。

本章介绍 Java SE 技术实现的 PetStore 宠物商店项目，所涉及的知识点：Java 面向对象、Lambda 表达式、Java Swing 技术、JDBC 技术和数据库相关等知识，其中还会用到方方面面的 Java 基础知识。

B.1 系统分析与设计

本节对 PetStore 宠物商店项目进行分析和设计，其中设计过程包括原型设计、数据库设计、架构设计和系统设计。

B.1.1 项目简介

PetStore 是 Sun（现在 Oracle）公司为了演示自己的 Java EE 技术而编写的一个基于 Web 的宠物店项目。如图 B-1 所示为项目启动页面，项目介绍网站是 http://www.oracle.com/technetwork/java/index-136650.html。

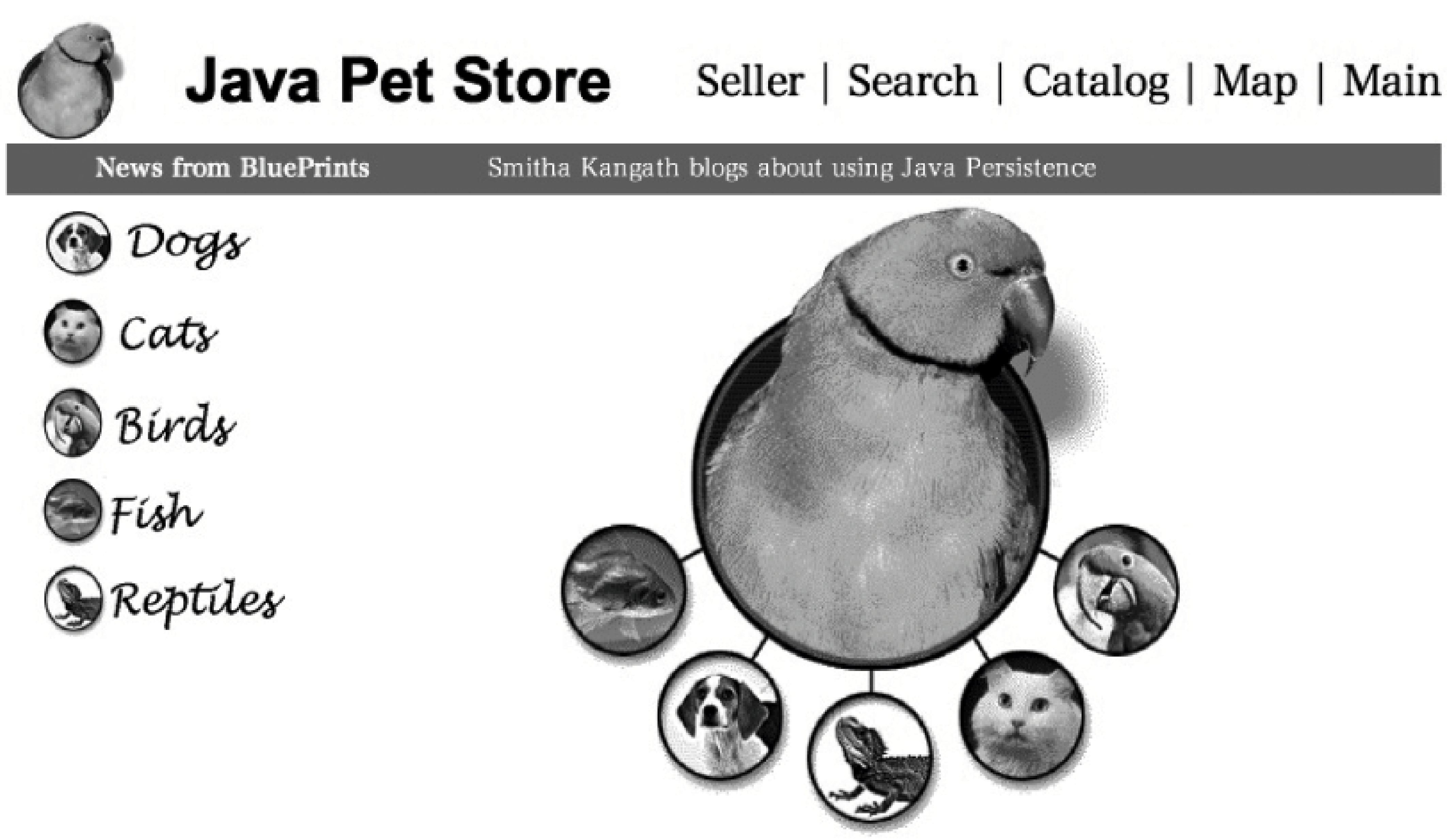


图 B-1 PetStore 项目启动页面

PetStore 是典型的电子商务项目，是现在很多电商平台的雏形。技术方面主要是 Java EE 技术，用户界面采用 Java Web 介绍实现。但本书介绍 Java SE 技术，不介绍 Java Web，所以本章的 PetStore 项目用户界面采用 Java Swing 技术实现。

B.1.2　需求分析

PetStore 宠物商店项目主要功能如下。

- 用户登录。
- 查询商品。
- 添加商品到购物车。
- 查看购物车。
- 下订单。
- 查看订单。

采用用例分析方法描述用例图如图 B-2 所示。

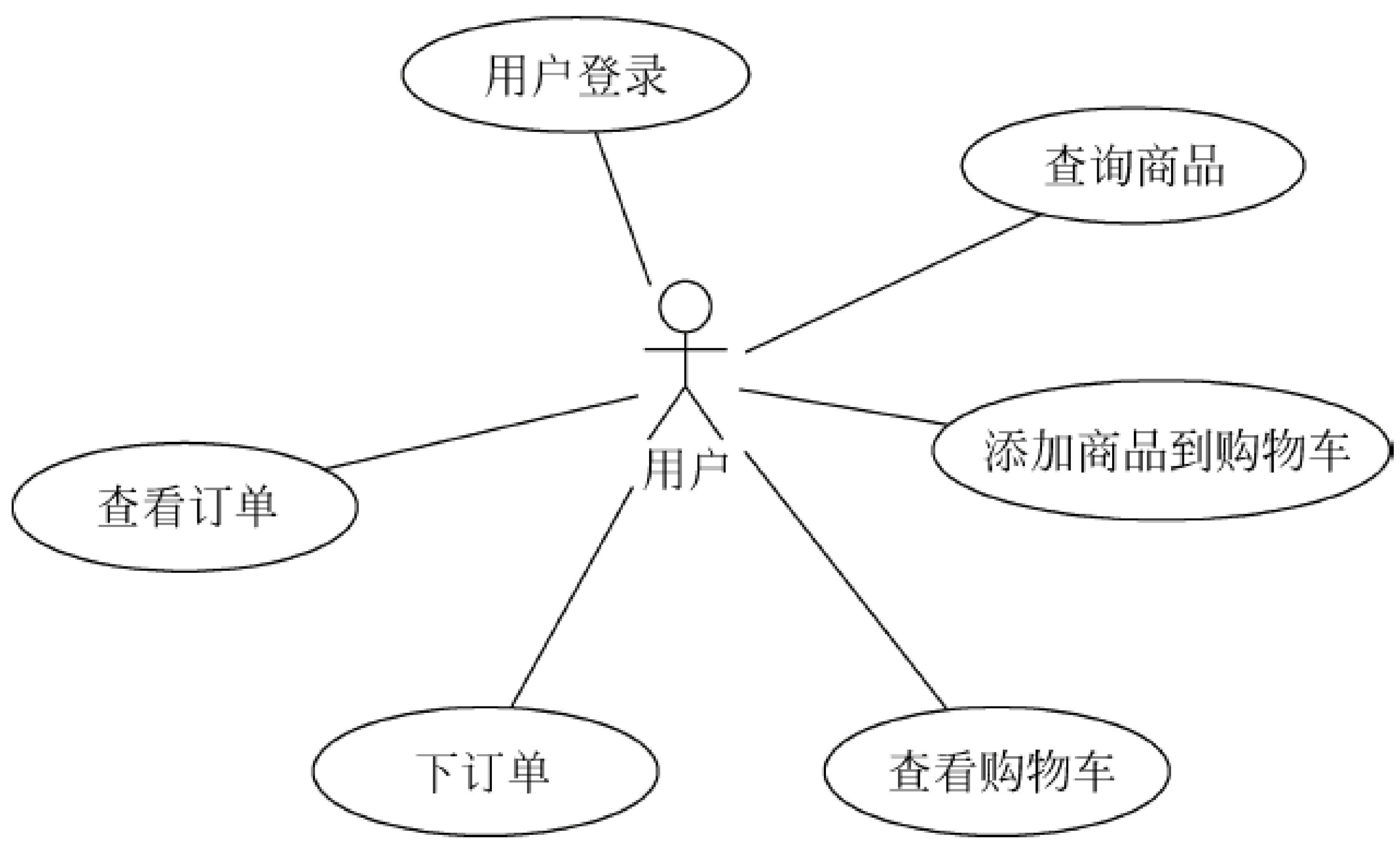


图 B-2　PetStore 宠物商店用例图

B.1.3　原型设计

原型设计草图对于开发人员、设计人员、测试人员、UI 设计人员以及用户都是非常重要的。PetStore 宠物商店项目原型设计图如图 B-3 所示。

B.1.4　数据库设计

Java 官方提供的 PetStore 宠物商店项目数据库设计比较复杂，这里根据如图 B-2 所示的用例图重新设计数据库。数据库设计模型如图 B-4 所示。

数据库设计模型中各个表说明如下。

1. 用户表

用户表(英文名 account)是 PetStore 宠物商店的注册用户，用户 Id(英文名 userid)是主键。用户表结构如表 B-1 所示。

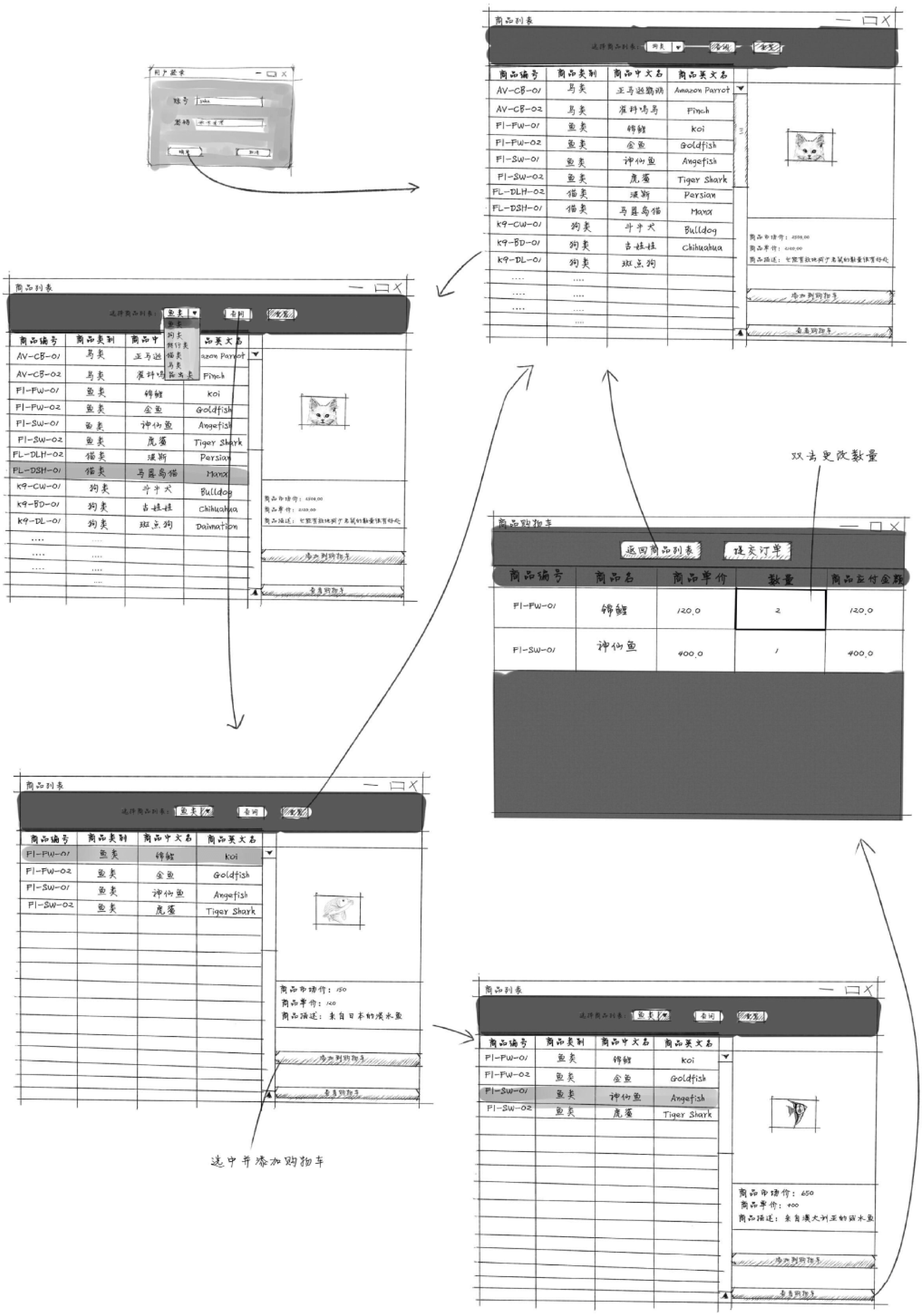


图 B-3　PetStore 宠物商店项目原型设计图

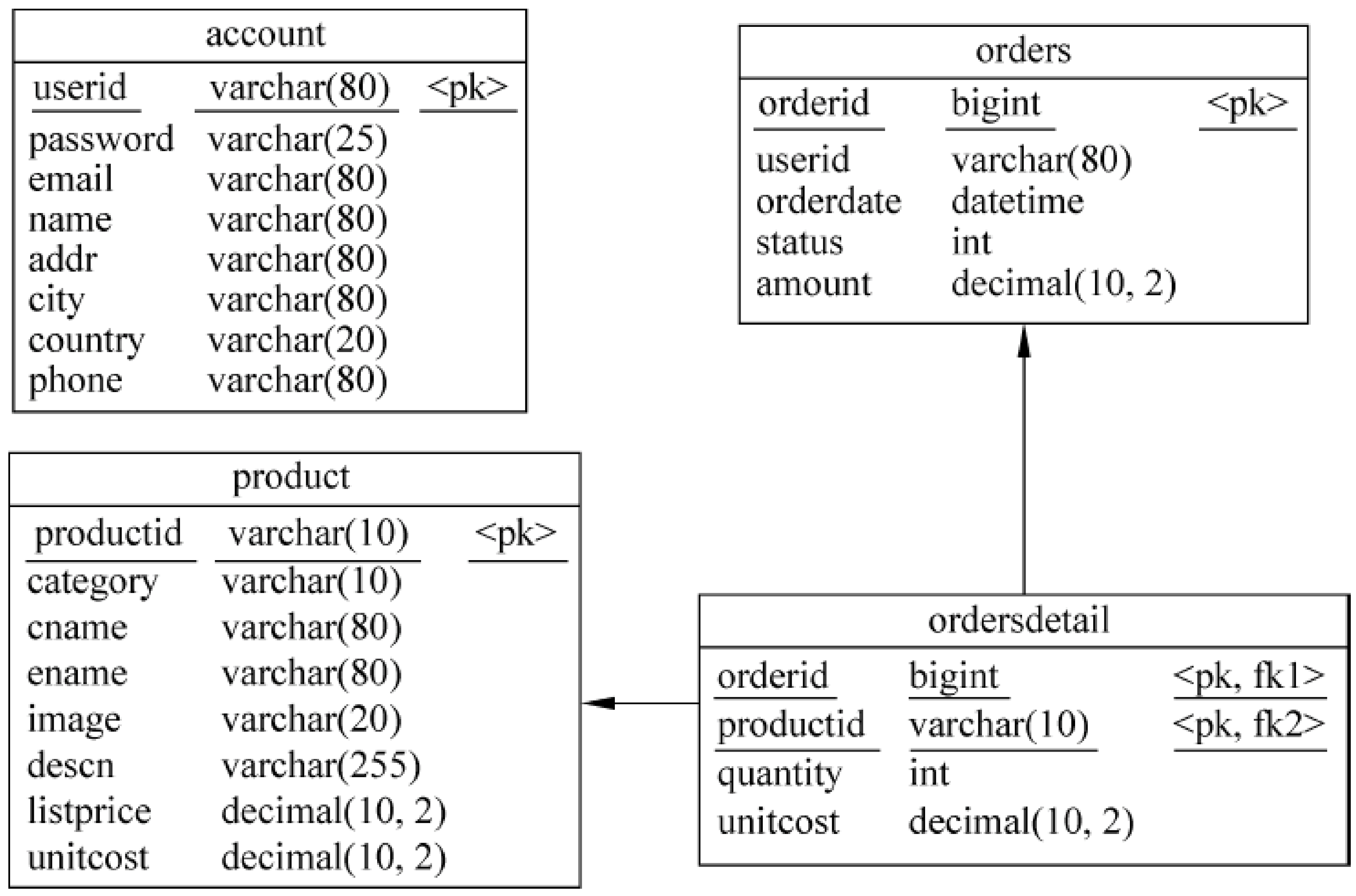


图 B-4　数据库设计模型

表 B-1　用户表

字段名	数据类型	长度	精度	主键	外键	备注
userid	varchar(80)	80	—	是	否	用户 Id
password	varchar(25)	25	—	否	否	用户密码
email	varchar(80)	80	—	否	否	用户 E-mail
name	varchar(80)	80	—	否	否	用户名
addr	varchar(80)	80	—	否	否	地址
city	varchar(80)	80	—	否	否	所在城市
country	varchar(20)	20	—	否	否	国家
phone	varchar(80)	80	—	否	否	电话号码

2. 商品表

商品表(英文名 product)是 PetStore 宠物商店所销售的商品(宠物),商品 Id(英文名 productid)是主键。商品表结构如表 B-2 所示。

表 B-2　商品表

字段名	数据类型	长度	精度	主键	外键	备注
productid	varchar(10)	10	—	是	否	商品 Id
category	varchar(10)	10	—	否	否	商品类别
cname	varchar(80)	80	—	否	否	商品中文名
ename	varchar(80)	80	—	否	否	商品英文名
image	varchar(20)	20	—	否	否	商品图片
descn	varchar(255)	255	—	否	否	商品描述
listprice	decimal(10,2)	10	2	否	否	商品市场价
unitcost	decimal(10,2)	10	2	否	否	商品单价

3. 订单表

订单表(英文名 orders)记录了用户每次购买商品所生成的订单信息,订单 Id(英文名

orderid)是主键。订单表结构如表 B-3 所示。

表 B-3　订单表

字　段　名	数据类型	长　　度	精　　度	主　　键	外　　键	备　　注
orderid	bigint		—	是	否	订单 Id
userid	varchar(80)	80	—	否	否	下订单的用户 Id
orderdate	datetime		—	否	否	下订单时间
status	int		—	否	否	订单付款状态,0 表示待付款;1 表示已付款
amount	decimal(10,2)	10	2	否	否	订单应付金额

4. 订单明细表

订单表中不能描述其中有哪些商品,购买商品的数量,购买时商品的单价等信息,这些信息被记录在订单明细表中。订单明细表(英文名 ordersdetail)记录了某个订单更加详细的详细,订单明细表主键是由 orderid 和 productid 两个字段联合而成,是一种联合主键。订单明细表结构如表 B-4 所示。

表 B-4　订单明细表

字　段　名	数据类型	长　　度	精　　度	主　　键	外　　键	备　　注
orderid	bigint		—	是	是	订单 Id
productid	varchar(10)	10	—	是	是	商品 Id
quantity	int		—	否	否	商品数量
unitcost	decimal(10,2)	10	2	否	否	商品单价

从图 B-4 所示的数据库设计模型中可以编写 DDL(数据定义)语句[①],使用这些语句可以很方便地创建和维护数据库中的表结构。

B.1.5　架构设计

无论庞大的企业级系统,还是手机上的应用,都应该有效地组织程序代码。这就需要设计。而架构设计就是系统的"骨架",它是源自于前人经验的总结和提炼,形成一种模式推而广之。但是遗憾的是本书的定位是初学者,本书并不是介绍架构设计方面的书。为了开发 PetStore 宠物商店项目需要,这里给出最简单的架构设计结果。

世界著名软件设计大师 Martin Fowler 在《企业应用架构模式》(英文名 *Patterns of Enterprise Application Architecture*)一书中提到,为了有效地组织代码,一个系统应该分为 3 个基本层,如图 B-5 所示。层(layer)是相似功能的类和接口的集合,层之间是松耦合的,层的内部是高内聚的。

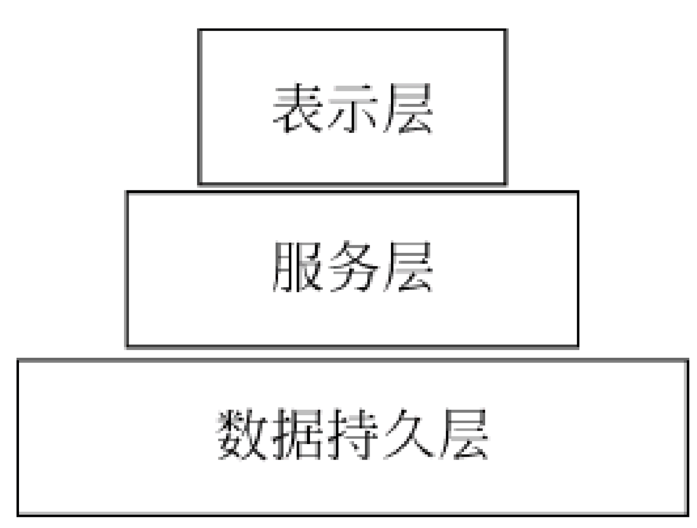


图 B-5　Martin Fowler 分层架构设计

- 表示层。用户与系统交互的组件集合。用户通过这一层向系统提交请求或发出指令,系统通过这一层接收用户

① 数据定义语句用于创建、删除和修改数据库对象,包括 DROP、CREATE、ALTER、GRANT、REVOKE 和 TRUNCATE 等语句。

请求或指令，待指令消化吸收后再调用下一层，接着将调用结果展现到这一层。表示层应该是轻薄的，不应该具有业务逻辑。

- 服务层。系统的核心业务处理层。负责接收表示层的指令和数据，待指令和数据消化吸收后，再进行组织业务逻辑的处理，并将结果返回给表示层。
- 数据持久层。用于访问持久化数据，持久化数据可以是保存在数据库、文件、其他系统或者网络中的数据。根据不同的数据来源，数据持久层会采用不同的技术。例如，如果数据保存到数据库中，则使用 JDBC 技术；如果数据保存到 JSON 文件中，则需要 I/O 流和 JSON 解码技术实现。

Martin Fowler 分层架构设计看起来像一个多层"蛋糕"，蛋糕师们在制作多层"蛋糕"的时候先做下层再做上层，最后做顶层。没有下层就没有上层，这叫作"上层依赖于下层"。为了降低松耦度，层之间还需要定义接口，通过接口隔离实现细节，上层调用者只关心接口，不关心下一层的实现细节。

Martin Fowler 分层架构是基本形式，在具体实现项目设计时，可能有所增加，也可能有所减少。本章实现的 PetStore 宠物商店项目，由于简化了需求，逻辑比较简单，可以不需要服务层，表示层可以直接访问数据持久层，如图 B-6 所示，表示层采用 Swing 技术实现，数据持久层采用 JDBC 技术实现。



图 B-6 PetStore 宠物商店项目架构设计

B.1.6 系统设计

系统设计是在具体架构下的设计实现，PetStore 宠物商店项目主要分为数据持久层和表示层。下面分别介绍一下它们的具体实现。

1. 数据持久层设计

数据持久层在具体实现时，会采用 DAO(数据访问对象)设计模式，数据库中每一个数据表对应一个 DAO 对象，每一个 DAO 对象中有访问数据表的 CRUD①4 类操作。

如图 B-7 所示为 PetStore 宠物商店项目的数据持久层类图。首先定义了 4 个 DAO 接口，这 4 个接口对应数据中的 4 个表，接口定义的方法是对数据库表的 CRUD 操作。

2. 表示层设计

主要使用 Swing 技术，每一个界面就是一个窗口对象。在表示层中各个窗口是依据原型设计而来的。PetStore 宠物商店项目表示层类如图 B-8 所示。其中有 3 个窗口类，即 LoginFrame(用户登录窗口)、CartFrame(购物车窗口)和 ProductListFrame(商品列表窗口)，它们有共同的父类 MyFrame，MyFrame 类是根据自己的项目情况进行的封装。从类图中可见，CartFrame 与 ProductListFrame 具有关联关系，CartFrame 包含一个对 ProductListFrame 的引用。

另外，CartFrame 与 ProductListFrame 会使用到表格，所以自定义了两个表模型 CartTableModel 和 ProductTableModel。

---

① CRUD 方法是访问数据的 4 个方法，即增加、删除、修改和查询：C 为 Create，表示增加数据；R 是 Read，表示查询数据；U 是 Update，表示修改数据；D 是 Delete，表示删除数据。

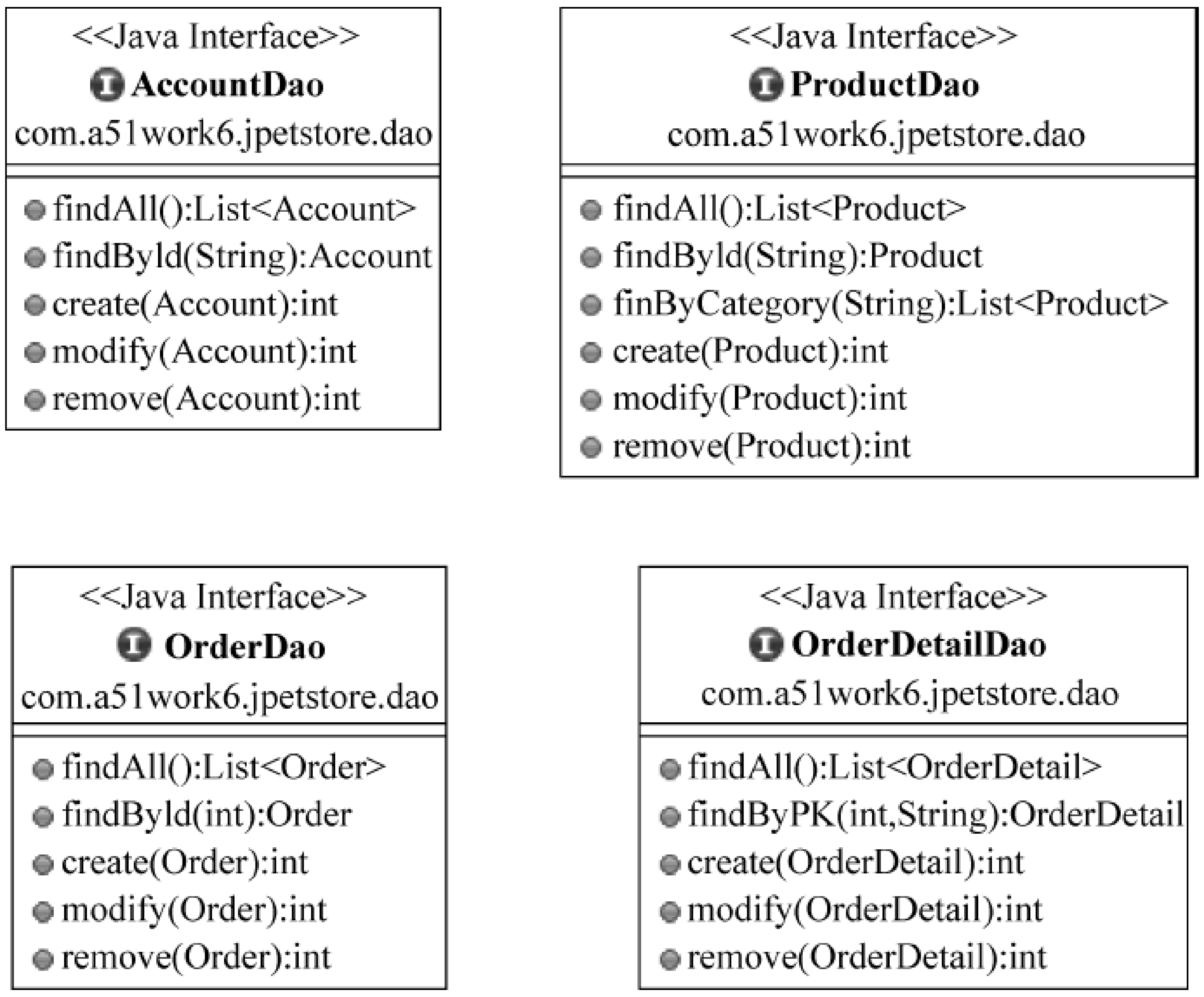


图 B-7 PetStore 宠物商店项目数据持久层类图

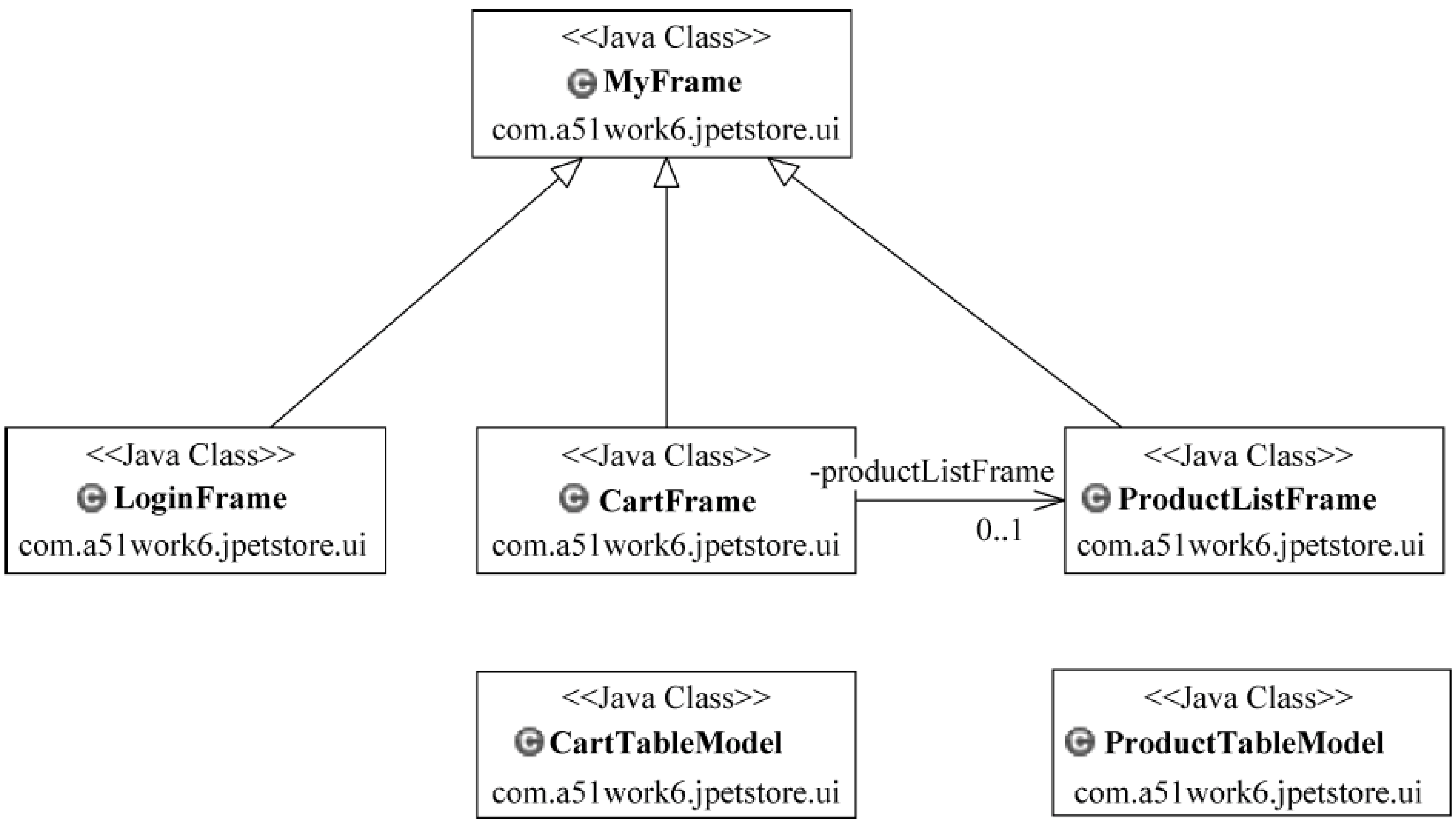


图 B-8 PetStore 宠物商店项目表示层类图

B.2 任务 1：创建数据库

在设计完成之后，在编写 Java 代码之前，应该创建数据库。

B.2.1 迭代 1.1：安装和配置 MySQL 数据库

首先应该为开发该项目，准备好数据库。本书推荐使用 MySQL 数据库，如果没有安装 MySQL 数据库，可以参考 A.2.1 节安装 MySQL 数据库。

B.2.2 迭代 1.2：编写数据库 DDL 脚本

按照图 B-4 所示的数据库设计模型编写数据库 DDL 脚本。当然，也可以通过一些工具生成 DDL 脚本，然后把这个脚本放在数据库中执行。下面是编写的 DDL 脚本：

```
/* 创建数据库 */
CREATE DATABASE IF NOT EXISTS petstore;

use petstore;

/* 用户表 */
CREATE TABLE IF NOT EXISTS account (
    userid varchar(80) not null,                /* 用户 Id */
    password varchar(25) not null,              /* 用户密码 */
    email varchar(80) not null,                 /* 用户 Email */
    name varchar(80) not null,                  /* 用户名 */
    addr varchar(80) not null,                  /* 地址 */
    city varchar(80) not null,                  /* 所在城市 */
    country varchar(20) not null,               /* 国家 */
    phone varchar(80) not null,                 /* 电话号码 */
PRIMARY KEY (userid));

/* 商品表 */
CREATE TABLE IF NOT EXISTS product (
    productid varchar(10) not null,             /* 商品 Id */
    category varchar(10) not null,              /* 商品类别 */
    cname varchar(80) null,                     /* 商品中文名 */
    ename varchar(80) null,                     /* 商品英文名 */
    image varchar(20) null,                     /* 商品图片 */
    descn varchar(255) null,                    /* 商品描述 */
    listprice decimal(10,2) null,               /* 商品市场价 */
    unitcost decimal(10,2) null,                /* 商品单价 */
PRIMARY KEY(productid));

/* 订单表 */
CREATE TABLE IF NOT EXISTS orders (
    orderid bigint not null,                    /* 订单 Id */
    userid varchar(80) not null,                /* 下订单的用户 Id */
    orderdate datetime not null,                /* 下订单时间 */
    status int not null default 0,              /* 订单付款状态,0 表示待付款,1 表示已付款 */
    amount decimal(10,2) not null,              /* 订单应付金额 */
PRIMARY KEY(orderid));

/* 订单明细表 */
CREATE TABLE IF NOT EXISTS ordersdetail (
    orderid bigint not null,                    /* 订单 Id */
    productid varchar(10) not null,             /* 商品 Id */
    quantity int not null,                      /* 商品数量 */
    unitcost decimal(10,2) null,                /* 商品单价 */
PRIMARY KEY(orderid, productid));
```

如果对于编写 DDL 脚本不熟悉，可以直接使用编写好的 jpetstore-mysql-schema-gbk.sql 脚本文件，文件位于 PetStore 项目下的 db 目录中。

B.2.3 迭代 1.3：插入初始数据到数据库

PetStore 宠物商店项目有一些初始的数据，这些初始数据在创建数据库之后插入。这些插入数据的语句如下：

```
use petstore;

    /* 用户表数据 */
    INSERT INTO account VALUES('j2ee','j2ee','yourname@yourdomain.com','关东升', '北京丰台区', '北京', '中国', '18811588888');
    INSERT INTO account VALUES('ACID','ACID','acid@yourdomain.com','Tony', '901 San Antonio Road', 'Palo Alto', 'USA', '555-555-5555');

    /* 商品表数据 */
    INSERT INTO product VALUES('FI-SW-01','鱼类','神仙鱼', 'Angelfish', 'fish1.jpg', '来自澳大利亚的咸水鱼', 650, 400);
    INSERT INTO product VALUES('FI-SW-02','鱼类','虎鲨', 'Tiger Shark', 'fish4.gif', '来自澳大利亚的咸水鱼', 850, 600);
    …
    INSERT INTO product VALUES('AV-CB-01','鸟类','亚马逊鹦鹉', 'Amazon Parrot', 'bird4.gif','寿命长达 75 年的大鸟', 3150, 3000);
    INSERT INTO product VALUES('AV-SB-02','鸟类','雀科鸣鸟', 'Finch', 'bird1.gif', '会唱歌的鸟儿', 150, 110);
```

如果不愿意自己编写插入数据的脚本文件，可以直接使用编写好的 jpetstore-mysql-dataload-gbk.sql 脚本文件，文件位于 PetStore 项目下的 db 目录中。

B.3 任务 2：初始化项目

本项目推荐使用 Eclipse 工具，所以首先参考 3.1 节创建一个 Eclipse 项目，项目名称为 PetStore。

B.3.1 任务 2.1：配置项目构建路径

PetStore 项目创建完成后，需要参考图 B-9，在 PetStore 项目根目录下创建普通文件夹 db。然后将 MySQL 数据库 JDBC 驱动程序 mysql-connector-java-5.xxx-bin.jar 复制到 db 目录，参考 A.3.2 节将驱动程序文件添加到项目的构建路径中。images 文件夹中内容是项目使用的图片。



图 B-9 PetStore 项目目录结构

B.3.2　任务 2.2：添加资源图片

项目中会用到很多资源图片，为了打包发布项目方便，这些图片最好放到 src 源文件夹下，Eclipse 会将该文件夹下所有文件一起复制到字节码文件夹中。参考图 B-9 在 src 文件夹下创建 images 文件夹，然后在本书配套资源中找到 images 中的图片，并将其复制到 Eclipse 项目的 images 文件夹中。

B.3.3　任务 2.3：添加包

参考图 B-9，在 src 文件夹中创建如下 4 个包。

- com.a51work6.jpetstore.ui。放置表示层组件。
- com.a51work6.jpetstore.domain。放置实体类。
- com.a51work6.jpetstore.dao。放置数据持久层组件中 DAO 接口。
- com.a51work6.jpetstore.dao.mysql。放置数据持久层组件中 DAO 接口具体实现类，mysql 说明是 MySQL 数据库 DAO 对象。该包中还放置了访问 MySQL 数据库的一些辅助类和配置文件。

B.4　任务 3：编写数据持久层代码

Eclipse 项目创建并初始化完成后，可以先编写数据持久层代码。

B.4.1　任务 3.1：编写实体类

无论数据库设计还是面向对象的架构设计都会使用“实体”。“实体”是系统中的“人”“事”“物”等名词，如用户、商品、订单和订单明细等。在数据库设计时它将演变为表，如用户表(account)、商品表(product)、订单表(orders)和订单明细表(ordersdetail)，在面向对象的架构设计时，实体将演变为“实体类”。如图 B-10 所示是 PetStore 宠物商店项目中的实体类。实体类属性与数据库表字段是相似的，事实上它们描述的是同一个事物，当然具有相同的属性，只是它们分别采用不同的设计理念，实体类采用对象模型，表采用关系模式。

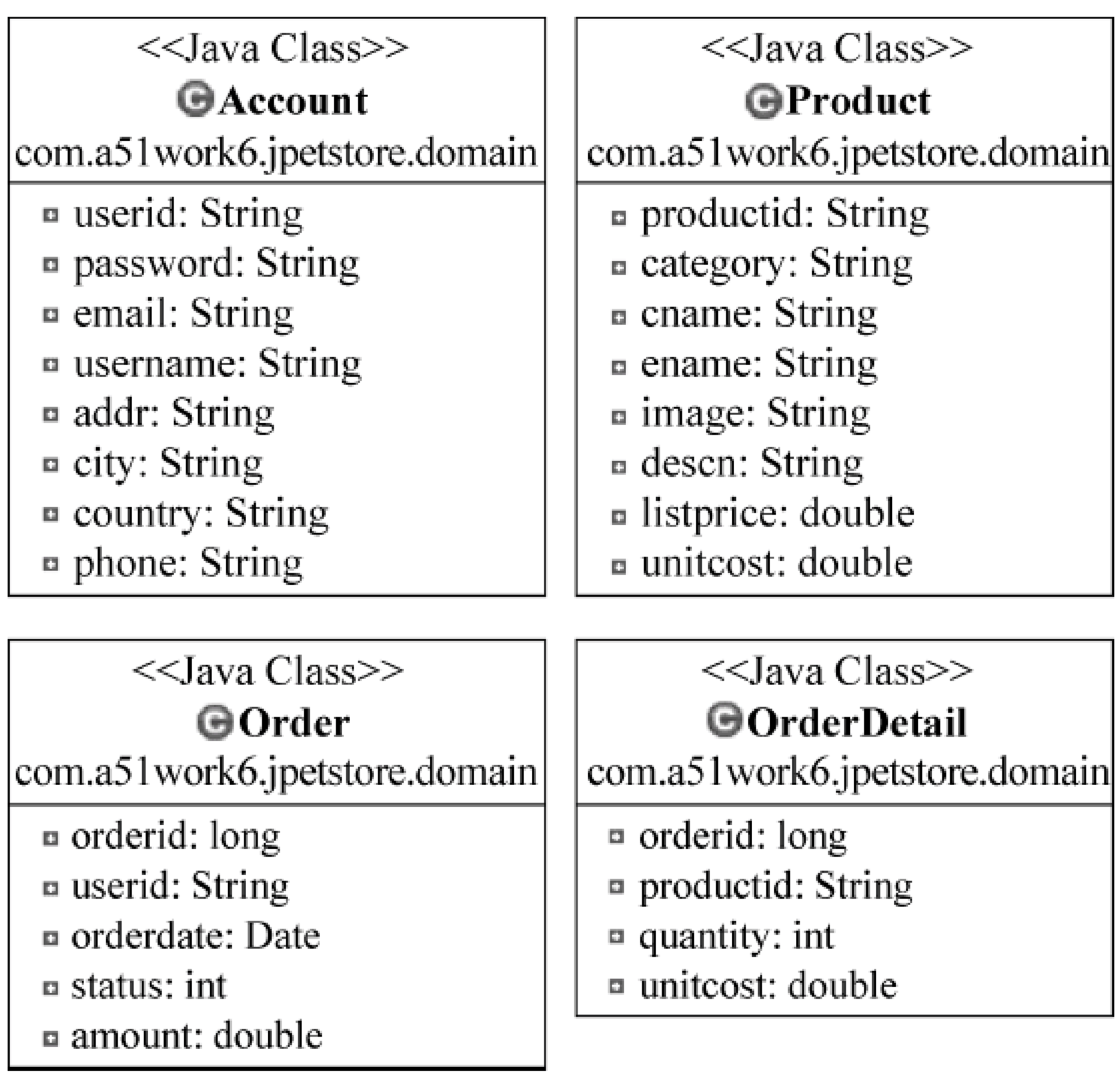


图 B-10　PetStore 宠物商店项目实体类图

订单明细实体类 OrderDetail 的代码如下：

```
//OrderDetail.java 文件
package com.a51work6.jpetstore.domain;

//订单明细
public class OrderDetail {

    private long orderid;                    //订单 Id
    private String productid;                //商品 Id
    private int quantity;                    //商品数量
    private double unitcost;                 //单价

    public long getOrderid() {
        return orderid;
    }

    public void setOrderid(long orderid) {
        this.orderid = orderid;
    }

    public double getUnitcost() {
        return unitcost;
    }

    public void setUnitcost(double unitcost) {
        this.unitcost = unitcost;
    }

    public String getProductid() {
        return productid;
    }

    public void setProductid(String productid) {
        this.productid = productid;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

}
```

从上述代码中可见，实体类结构很简单，主要是一个私有属性，以及对这些属性方法的公有 Getter 和 Setter 方法。在使用 Eclipse 编程时只需要编写那些私有属性即可，然后通过 Eclipse 工具生成 Getter 和 Setter 方法，具体步骤参考 23.6 节。

订单实体类 Order 的代码如下：

```
//Order.java 文件
package com.a51work6.jpetstore.domain;

import java.util.Date;

public class Order {

    private long orderid;           //订单 Id
    private String userid;          //下订单的用户 Id
    private Date orderdate;         //下订单时间
    private int status;             //订单付款状态,0 表示待付款,1 表示已付款
    private double amount;          //订单应付金额

    //省略 Setter 和 Getter 方法

}
```

用户实体类 Account 的代码如下：

```
//Account.java 文件
package com.a51work6.jpetstore.domain;

public class Account {

    /* 私有成员变量 */
    private String userid;          //用户 Id
    private String password;        //用户密码
    private String email;           //用户 E-mail
    private String username;        //用户名
    private String addr;            //地址
    private String city;            //所在城市
    private String country;         //国家
    private String phone;           //电话号码

    //省略 Setter 和 Getter 方法

}
```

商品实体类 Product 的代码如下：

```
//Product.java 文件
package com.a51work6.jpetstore.domain;

public class Product {

    private String productid;           //商品 Id
    private String category;            //商品类别
    private String cname;               //商品中文名
```

```
    private String ename;                    //商品英文名
    private String image;                    //商品图片
    private String descn;                    //商品描述
    private double listprice;                //商品市场价
    private double unitcost;                 //商品单价

    //省略 Setter 和 Getter 方法

}
```

B.4.2 迭代 3.2：编写 DAO 类

编写 DAO 类就没有实体类那么简单了，数据持久层开发的工作量主要是 DAO 类。如图 B-11 所示是 DAO 实现类图。

1. 用户管理 DAO

用户管理 AccountDao 实现类 AccountDaoImp 代码如下：

```
//AccountDaoImp.java 文件
package com.a51work6.jpetstore.dao.mysql;
…
public class AccountDaoImp implements AccountDao {

    @Override
    public List<Account> findAll() {
        //TODO 自动生成的方法存根
        return null;
    }

    @Override
    public Account findById(String userid) {

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        Account account = null;
        try {
            //2. 创建数据库连接
            conn = DBHelper.getConnection();
            //3. 创建语句对象
            String sql = "select userid,password,email,name,addr,city, country,phone" +
" from account where userid = ?";

            pstmt = conn.prepareStatement(sql);
            //4. 绑定参数
            pstmt.setString(1, userid);
            //5. 执行查询(R)
            rs = pstmt.executeQuery();
            //6. 遍历结果集
```

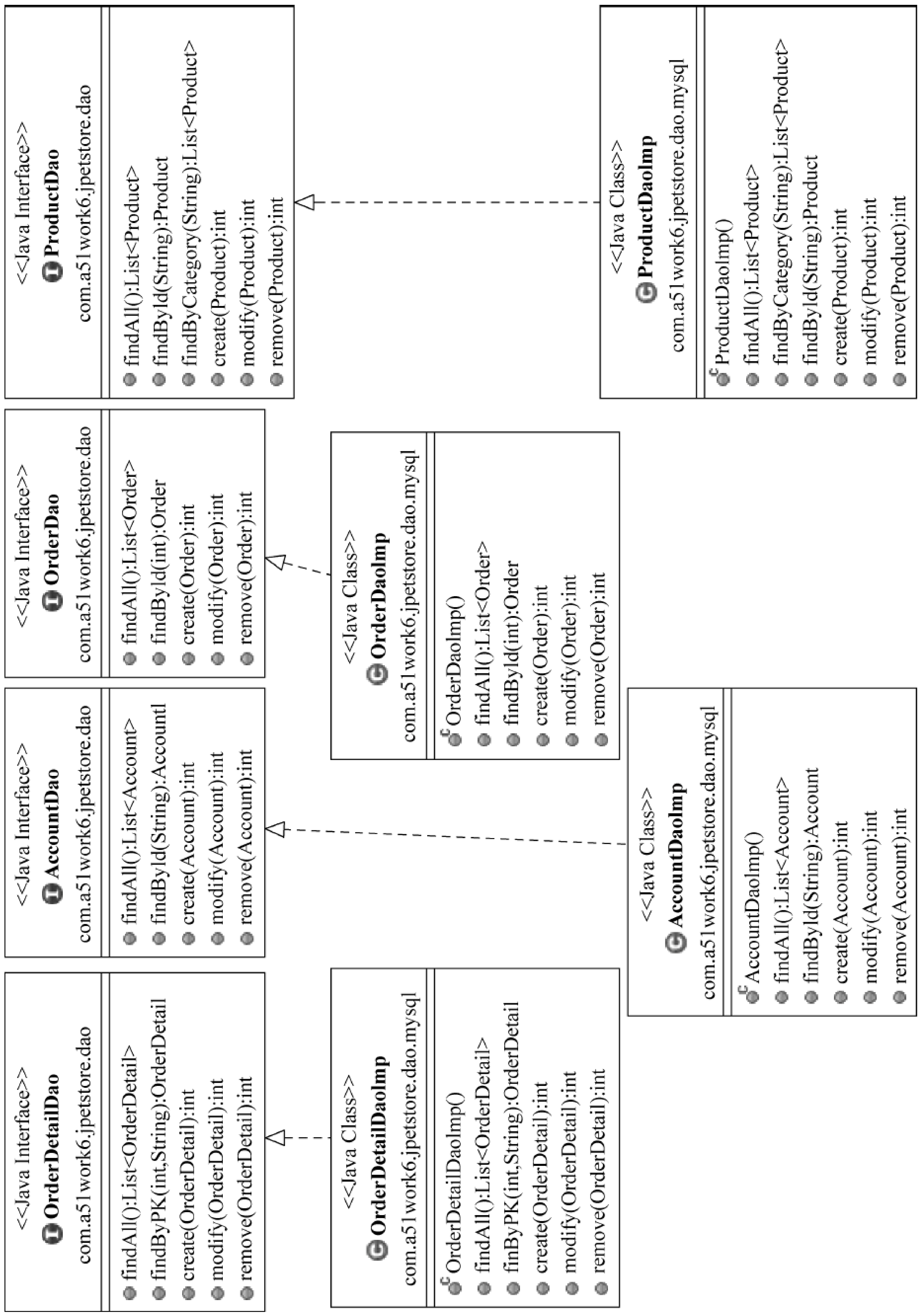


图 B-11 DAO实现类图

```
            if (rs.next()) {
                account = new Account();

                account.setUserid(rs.getString("userid"));
                account.setPassword(rs.getString("password"));
                account.setEmail(rs.getString("email"));
                account.setUsername(rs.getString("name"));
                account.setAddr(rs.getString("addr"));
                account.setUserid(rs.getString("userid"));
                account.setCity(rs.getString("city"));
                account.setCountry(rs.getString("country"));
                account.setPhone(rs.getString("phone"));

                return account;
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {                                  //释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
            try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return null;
    }

    @Override
    public int create(Account account) {
        //TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int modify(Account account) {
```

```
        //TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int remove(Account account) {
        //TODO 自动生成的方法存根
        return 0;
    }

}
```

AccountDao 接口中定义了 5 个抽象方法。但在本项目中只需要实现 findById()方法。具体代码不再赘述。

2. 商品管理 DAO

商品管理 ProductDao 实现类 ProductDaoImp 代码如下：

```
//ProductDaoImp.java 文件
package com.a51work6.jpetstore.dao.mysql;
…
//商品管理 DAO
public class ProductDaoImp implements ProductDao {

    @Override
    public List<Product> findAll() {

        String sql = "select productid,category,cname,ename,image,"
                + "listprice,unitcost,descn from product";

        List<Product> products = new ArrayList<Product>();

        try (
                //2. 创建数据库连接
                Connection conn = DBHelper.getConnection();
                //3. 创建语句对象
                  String sql = "select productid, category, cname, ename, image, listprice,
unitcost,descn "
                + "from product where category = ?";

                pstmt = conn.prepareStatement(sql);
                //4. 绑定参数
                pstmt.setString(1, category);

                //5. 执行查询(R)
                ResultSet rs = pstmt.executeQuery()) {

            //6. 遍历结果集
            while (rs.next()) {
                Product p = new Product();
```

```
                p.setProductid(rs.getString("productid"));
                p.setCategory(rs.getString("category"));
                p.setCname(rs.getString("cname"));
                p.setEname(rs.getString("ename"));
                p.setImage(rs.getString("image"));
                p.setListprice(rs.getDouble("listprice"));
                p.setUnitcost(rs.getDouble("unitcost"));
                p.setDescn(rs.getString("descn"));

                products.add(p);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
        return products;
    }

    @Override
    public List<Product> findByCategory(String category) {

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        List<Product> products = new ArrayList<Product>();

        try {
            //2. 创建数据库连接
            conn = DBHelper.getConnection();
            //3. 创建语句对象
            String sql = "select productid,category,cname,ename,image, listprice,unitcost,
descn " + "from product where category = ?";

            pstmt = conn.prepareStatement(sql);
            //4. 绑定参数
            pstmt.setString(1, category);
            //5. 执行查询(R)
            rs = pstmt.executeQuery();
            //6. 遍历结果集
            while (rs.next()) {
                Product p = new Product();
                p.setProductid(rs.getString("productid"));
                p.setCategory(rs.getString("category"));
                p.setCname(rs.getString("cname"));
                p.setEname(rs.getString("ename"));
                p.setImage(rs.getString("image"));
                p.setListprice(rs.getDouble("listprice"));
                p.setUnitcost(rs.getDouble("unitcost"));
                p.setDescn(rs.getString("descn"));
```

```
                products.add(p);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {                                  //释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return products;
    }

    @Override
    public Product findById(String productid) {

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            //2. 创建数据库连接
            conn = DBHelper.getConnection();
            //3. 创建语句对象
            String sql = "select productid,category,cname,ename,image, listprice,unitcost,
descn " + "from product where productid = ?";

            pstmt = conn.prepareStatement(sql);
            //4. 绑定参数
            pstmt.setString(1, productid);
            //5. 执行查询(R)
            rs = pstmt.executeQuery();
            //6. 遍历结果集
            if (rs.next()) {
```

```
                Product p = new Product();
                p.setProductid(rs.getString("productid"));
                p.setCategory(rs.getString("category"));
                p.setCname(rs.getString("cname"));
                p.setEname(rs.getString("ename"));
                p.setImage(rs.getString("image"));
                p.setListprice(rs.getDouble("listprice"));
                p.setUnitcost(rs.getDouble("unitcost"));
                p.setDescn(rs.getString("descn"));

                return p;
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {                              //释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return null;
    }

    @Override
    public int create(Product product) {
        //TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int modify(Product product) {
        //TODO 自动生成的方法存根
        return 0;
    }
```

```
    @Override
    public int remove(Product product) {
        //TODO 自动生成的方法存根
        return 0;
    }

}
```

ProductDao 接口中定义了 6 个抽象方法，但在本项目中只需要实现 findAll()、findByCategory()和 findById()方法。

3. 订单管理 DAO

订单管理 OrderDao 实现类 OrderDaoImp 代码如下：

```
//OrderDaoImp.java 文件
package com.a51work6.jpetstore.dao.mysql;
…
//订单管理 DAO
public class OrderDaoImp implements OrderDao {

    @Override
    public List<Order> findAll() {

        String sql = "select orderid,userid,orderdate from product";
        List<Order> orderList = new ArrayList<Order>();

        try (
                //2. 创建数据库连接
                Connection conn = DBHelper.getConnection();
                //3. 创建语句对象
                PreparedStatement pstmt = conn.prepareStatement(sql);
                //5. 执行查询(R)
                ResultSet rs = pstmt.executeQuery()) {

            //6. 遍历结果集
            while (rs.next()) {
                Order order = new Order();
                order.setOrderid(rs.getInt("orderid"));
                order.setUserid(rs.getString("userid"));
                order.setOrderdate(rs.getDate("orderdate"));

                orderList.add(order);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
        return orderList;
    }
```

```
    @Override
    public Order findById(int orderid) {
        //TODO 自动生成的方法存根
        return null;
    }

    @Override
    public int create(Order order) {

        try ( //2. 创建数据库连接
                Connection conn = DBHelper.getConnection();
                //3. 创建语句对象
                PreparedStatement pstmt = conn.prepareStatement(
                        "insert into orders (orderid,userid,orderdate, status,amount)" +
"values(?,?,?,?,?)")) {

            //4. 绑定参数
            pstmt.setLong(1, order.getOrderid());
            pstmt.setString(2, order.getUserid());
            java.util.Date date = order.getOrderdate();
            //pstmt.setDate(3, new java.sql.Date(date.getTime()));
            pstmt.setTimestamp(3, new java.sql.Timestamp(date.getTime()));
            pstmt.setInt(4, order.getStatus());
            pstmt.setDouble(5, order.getAmount());

            //5. 执行修改(C、U、D)
            int affectedRows = pstmt.executeUpdate();
            System.out.printf("成功插入%d条数据.\n", affectedRows);

        } catch (SQLException e) {
            return -1;
        }

        return 0;
    }

    @Override
    public int modify(Order order) {
        //TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int remove(Order order) {
        //TODO 自动生成的方法存根
        return 0;
    }

}
```

OrderDao 接口中定义了 5 个抽象方法，但在本项目中只需要实现 findAll() 和 create() 方法。

4．订单明细管理 DAO

订单明细管理 OrderDetailDao 实现类 OrderDetailDaoImp 代码如下：

```
//OrderDetailDaoImp.java 文件
package com.a51work6.jpetstore.dao.mysql;
…
//订单明细管理 DAO
public class OrderDetailDaoImp implements OrderDetailDao {

    @Override
    public List<OrderDetail> findAll() {
        //TODO 自动生成的方法存根
        return null;
    }

    @Override
    public OrderDetail findByPK(int orderid, String productid) {

        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        OrderDetail orderDetail = null;

        try {
            //2. 创建数据库连接
            conn = DBHelper.getConnection();
            //3. 创建语句对象
            String sql = "select orderid,productid,quantity,unitprice "
                    + "from ordersdetail where orderid = ? and productid = ?";

            pstmt = conn.prepareStatement(sql);
            //4. 绑定参数
            pstmt.setInt(1, orderid);
            pstmt.setString(2, productid);

            //5. 执行查询(R)
            rs = pstmt.executeQuery();
            //6. 遍历结果集
            if (rs.next()) {

                orderDetail = new OrderDetail();
                orderDetail.setOrderid(rs.getInt("orderid"));
                orderDetail.setProductid(rs.getString("productid"));
                orderDetail.setQuantity(rs.getInt("quantity"));
                orderDetail.setUnitcost(rs.getDouble("unitcost"));

                return orderDetail;
```

```
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {                              //释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                }
            }
            if (pstmt != null) {
                try {
                    pstmt.close();
                } catch (SQLException e) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                }
            }
        }

        return null;
    }

    @Override
    public int create(OrderDetail orderDetail) {
        try ( //2. 创建数据库连接
                Connection conn = DBHelper.getConnection();
                //3. 创建语句对象

                PreparedStatement pstmt = conn
                        .prepareStatement("insert into ordersdetail "
                                + "(orderid, productid,quantity,unitcost)
values (?,?,?,?)")) {

            //4. 绑定参数
            pstmt.setLong(1, orderDetail.getOrderid());
            pstmt.setString(2, orderDetail.getProductid());
            pstmt.setInt(3, orderDetail.getQuantity());
            pstmt.setDouble(4, orderDetail.getUnitcost());

            //5. 执行修改(C、U、D)
            int affectedRows = pstmt.executeUpdate();
            System.out.printf("成功插入%d条数据.\n", affectedRows);

        } catch (SQLException e) {
```

```
            return -1;
        }
        return 0;
    }

    @Override
    public int modify(OrderDetail orderDetail) {
        //TODO 自动生成的方法存根
        return 0;
    }

    @Override
    public int remove(OrderDetail orderDetail) {
        //TODO 自动生成的方法存根
        return 0;
    }

}
```

OrderDetailDao 接口中定义了 5 个抽象方法。但在本项目中只需要实现 findByPK()和 create()方法。

B.4.3 迭代 3.3：数据库帮助类 DBHelper

数据库帮助类 DBHelper 可以进行 JDBC 驱动程序加载以及获得数据库连接。具体实现代码如下：

```
//DBHelper.java 文件
package com.a51work6.jpetstore.dao.mysql;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

//数据库辅助类
public class DBHelper {

    //连接数据库 url
    static String url;
    //创建 Properties 对象
    static Properties info = new Properties();

    //1. 驱动程序加载
    static {                                                    ①
        //获得属性文件输入流
        InputStream input = DBHelper.class.getClassLoader()
```

```
                    .getResourceAsStream("com/a51work6/jpetstore/dao/mysql/config.properties");

        try {
            //加载属性文件内容到 Properties 对象
            info.load(input);
            //从属性文件中取出 url
            url = info.getProperty("url");
            //从属性文件中取出 driver
            String driverClassName = info.getProperty("driver");
            Class.forName(driverClassName);
            System.out.println("驱动程序加载成功…");
        } catch (ClassNotFoundException e) {
            System.out.println("驱动程序加载失败…");
        } catch (IOException e) {
            System.out.println("加载属性文件失败…");
        }
    }                                                                   ②

    //获得数据库连接
    public static Connection getConnection() throws SQLException {      ③
        //创建数据库连接
        Connection conn = DriverManager.getConnection(url, info);
        return conn;
    }

}
```

上述代码第①～②行通过静态代码库加载数据库驱动程序，并且在静态代码块中读取配置文件 config.properties 信息，该配置文件位于 com.a51work6.jpetstore.dao.mysql 包中，内容如下：

```
driver = com.mysql.jdbc.Driver
url = jdbc:mysql://localhost:3306/petstore
user = root
password = 123456
useSSL = false
verifyServerCertificate = false
```

代码第③行提供了获得数据库的连接方法，这是一个静态方法，使用起来比较方便。

B.5 任务 4：编写表示层代码

从客观上讲，表示层开发的工作量是很大的，有很多细节工作需要完成。

B.5.1 迭代 4.1：编写启动类

Java SE 应用程序需要有一个具有 main 主方法的类，它是项目启动类。代码如下：

```
//MainApp.java 文件
package com.a51work6.jpetstore.ui;

import com.a51work6.jpetstore.domain.Account;

//启动类
public class MainApp {

    //用户登录成功后,保存当前用户信息
    public static Account accout;                                  ①

    public static void main(String[] args) {

        LoginFrame frame = new LoginFrame();
        frame.setVisible(true);
    }

}
```

在 main 主方法中实例化用户登录窗口——LoginFrame 类。另外，代码第①行声明静态变量 accout，当用户登录成功后 accout 用来保存用户信息。静态变量 accout 可以在其他类中方便访问。这是为了模拟 Web 应用开发中的会话(Session)对象，等用户打开浏览器，登录 Web 系统后，服务器端会将用户信息保存到会话对象中。

B.5.2　迭代 4.2：编写自定义窗口类——MyFrame

由于 Swing 提供的 JFrame 类启动窗口后默认位于屏幕的左上角，而本项目中所有的窗口都是屏幕居中的，因此自定义了窗口类 MyFrame。MyFrame 代码如下：

```
//MyFrame.java 文件
package com.a51work6.jpetstore.ui;

import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;

//这是一个屏幕居中的自定义窗口
public class MyFrame extends JFrame {

    //获得当前屏幕的宽
    private double screenWidth = Toolkit.getDefaultToolkit().getScreenSize().getWidth();      ①
    //获得当前屏幕的高
    private double screenHeight = Toolkit.getDefaultToolkit().getScreenSize().getHeight();    ②

    public MyFrame(String title, int width, int height) {
```

```
            super(title);

            //设置窗口大小
            setSize(width, height);
            //计算窗口位于屏幕中心的坐标
            int x = (int) (screenWidth - width) / 2;                ③
            int y = (int) (screenHeight - height) / 2;              ④
            //设置窗口位于屏幕中心
            setLocation(x, y);

            //注册窗口事件
            addWindowListener(new WindowAdapter() {                 ⑤
                //单击窗口关闭按钮时调用
                public void windowClosing(WindowEvent e) {
                    //退出系统
                    System.exit(0);
                }
            });
        }
    }
```

上述代码第①行和第②行是获取当前屏幕的宽和高，具体的计算过程见代码第③行和第④行，具体的原理在23.5.7节已经介绍过，这里不再赘述。

另外，代码第⑤行注册窗口事件，当用户单击窗口的关闭按钮时调用System.exit(0)语句退出系统，继承MyFrame类的所有窗口都可以单击关闭按钮时退出系统。

B.5.3 迭代4.3：用户登录窗口

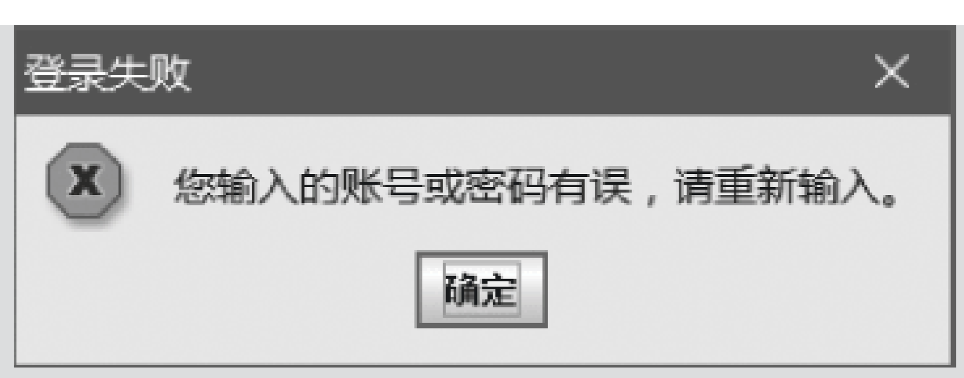
MainApp类运行时会启动用户登录窗口，如图B-12所示，其中有一个文本框、一个密码框和两个按钮。用户输入账号和密码后，单击“确定”按钮，如果输入的账号和密码正确，则登录成功进入商品列表窗口；如果输入的账号或密码不正确，则弹出如图B-13所示的对话框。



图B-12 用户登录窗口

用户登录窗口LoginFrame代码如下：

```
//LoginFrame.java文件
package com.a51work6.jpetstore.ui;
…
//用户登录窗口
public class LoginFrame extends MyFrame {
```



图B-13 用户登录失败提示

```
    private JTextField txtAccountId;
    private JPasswordField txtPassword;

    public LoginFrame() {
        super("用户登录", 400, 230);
        //设置布局管理为绝对布局
        getContentPane().setLayout(null);

        JLabel label1 = new JLabel();
        label1.setHorizontalAlignment(SwingConstants.RIGHT);
        label1.setBounds(51, 33, 83, 30);
        getContentPane().add(label1);
        label1.setText("账号：");
        label1.setFont(new Font("微软雅黑", Font.PLAIN, 15));

        txtAccountId = new JTextField(10);
        txtAccountId.setText();
        txtAccountId.setBounds(158, 33, 157, 30);
        txtAccountId.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        getContentPane().add(txtAccountId);

        JLabel label2 = new JLabel();
        label2.setText("密码：");
        label2.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        label2.setHorizontalAlignment(SwingConstants.RIGHT);
        label2.setBounds(51, 85, 83, 30);
        getContentPane().add(label2);

        txtPassword = new JPasswordField(10);
        txtPassword.setText();
        txtPassword.setBounds(158, 85, 157, 30);
        getContentPane().add(txtPassword);

        JButton btnOk = new JButton();
        btnOk.setText("确定");
        btnOk.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        btnOk.setBounds(61, 140, 100, 30);
        getContentPane().add(btnOk);

        JButton btnCancel = new JButton();
        btnCancel.setText("取消");
        btnCancel.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        btnCancel.setBounds(225, 140, 100, 30);
        getContentPane().add(btnCancel);

        //注册 btnOk 的 ActionEvent 事件监听器
        btnOk.addActionListener(e -> {                                    ①
            AccountDao accountDao = new AccountDaoImp();                  ②
            Account account = accountDao.findById(txtAccountId
.getText());                                                              ③
```

```
            String passwordText = new String(txtPassword.getPassword());      ④
            if (account != null && passwordText.equals(account
    .get Password())) {                                                        ⑤
                System.out.println("登录成功.");
                ProductListFrame form = new ProductListFrame();
                form.setVisible(true);
                setVisible(false);
                //用户登录成功后,将用户信息保存到 MainApp.accout 静态变量中
                MainApp.accout = account;
            } else {
                JLabel label = new JLabel("您输入的账号或密码有误,请重新输入.");
                label.setFont(new Font("微软雅黑", Font.PLAIN, 15));
                JOptionPane.showMessageDialog(null, label, "登录失败",
                        JOptionPane.ERROR_MESSAGE);                             ⑥
            }

        });

        //注册 btnCancel 的 ActionEvent 事件监听器
        btnCancel.addActionListener(e -> {                                     ⑦
            //退出系统
            System.exit(0);
        });
    }
}
```

上述代码第①行是用户单击“确定”按钮调用代码块。代码第②行创建 AccountDaoImp 对象,代码第③行通过 DAO 对象调用 findById()方法,该方法通过用户账号查询用户信息。代码第④行从密码框中取出密码。代码第⑤行比较窗口界面中的密码与从数据库中查询的密码是否一致,如果一致则登录成功;否则登录失败,失败时弹出对话框。代码第⑥行中的 JOptionPane. showMessageDialog 方法可以弹出对话框。JOptionPane 类还有如下类似的静态方法。

- showConfirmDialog:弹出确认框,可以是 Yes、No、Ok 和 Cancel 等按钮。
- showInputDialog:弹出提示输入对话框。
- showMessageDialog:弹出消息提示对话框。

代码第⑥行的 showMessageDialog 方法第一个参数是设置对话框所在的窗口,如果是当前窗口,则设置为 null;第二个参数是要显示的消息;第三个参数是对话框标题;第四个参数是要显示的消息类型,这些类型有 ERROR_MESSAGE、INFORMATION_MESSAGE、WARNING_MESSAGE、QUESTION_MESSAGE 或 PLAIN_MESSAGE,其中 ERROR_ MESSAGE 是错误消息类型。

B.5.4 迭代 4.4:商品列表窗口

登录成功后会进入商品列表窗口,如图 B-14 所示。商品列表窗口是分栏显示的,左栏是商品列表,右栏是商品明细信息。商品列表窗口是 PetStore 项目的最核心窗口,在该窗口可进行的操作如下。

• 查看商品信息。当在左栏的表格中选择某一个商品时，右栏会显示该商品的详细信息。



图 B-14 商品列表窗口

• 选择商品类型进行查询。用户可以选择商品类型，单击“查询”按钮根据商品类型进行查询。如图 B-15 所示为选中“鱼类”商品类型时查询结果。



图 B-15 查询商品列表

• 重置查询。根据商品类型查询后，如果想返回查询之前的状态，可以单击“重置”按钮重置商品列表，回到如图 B-14 所示界面。
• 添加商品到购物车。用户在商品列表中选中商品后，可以单击“添加到购物车”按钮，将选中的商品添加到购物车中。注意用户每单击一次增加一次该商品的数量到购物车。
• 查看购物车。用户单击“查看购物车”按钮后窗口会跳转到购物车窗口。

商品列表窗口 ProductListFrame 代码如下：

```
//ProductListFrame.java 文件
package com.a51work6.jpetstore.ui;
…

//商品列表窗口
public class ProductListFrame extends MyFrame {

    private JTable table;
    private JLabel lblImage;
    private JLabel lblListprice;
    private JLabel lblDescn;
    private JLabel lblUnitcost;

    //商品列表集合
    private List<Product> products = null;
    //创建商品 Dao 对象
    private ProductDao dao = new ProductDaoImp();

    //购物车,键是选择的商品 Id,值是商品的数量
    private Map<String, Integer> cart = new HashMap<String, Integer>();
    //选择的商品索引
    private int selectedRow = -1;

    public ProductListFrame() {

        super("商品列表", 1000, 700);
        //查询所有商品
        products = dao.findAll();                                          ①

        //添加顶部搜索面板
        getContentPane().add(getSearchPanel(), BorderLayout.NORTH);

        //创建分栏面板
        JSplitPane splitPane = new JSplitPane();
        //设置指定分隔条位置,从窗格的左边到分隔条的左边
        splitPane.setDividerLocation(600);
        //设置左侧面板
        splitPane.setLeftComponent(getLeftPanel());
        //设置右侧面板
        splitPane.setRightComponent(getRightPanel());
```

```
        //把分栏面板添加到内容面板
        getContentPane().add(splitPane, BorderLayout.CENTER);
    }

    //初始化搜索面板
    private JPanel getSearchPanel() {

        JPanel searchPanel = new JPanel();
        FlowLayout flowLayout = (FlowLayout) searchPanel.getLayout();
        flowLayout.setVgap(20);
        flowLayout.setHgap(40);

        JLabel lbl = new JLabel("选择商品类别：");
        lbl.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(lbl);

        String[] categorys = { "鱼类", "狗类", "爬行类", "猫类", "鸟类" };
        JComboBox comboBox = new JComboBox(categorys);
        comboBox.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(comboBox);

        JButton btnGo = new JButton("查询");
        btnGo.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(btnGo);

        JButton btnReset = new JButton("重置");
        btnReset.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        searchPanel.add(btnReset);

        //注册"查询"按钮的 ActionEvent 事件监听器
        btnGo.addActionListener(e -> {                                  ②
            //所选择的类别
            String category = (String) comboBox.getSelectedItem();
            //按照类别进行查询
            products = dao.findByCategory(category);
            TableModel model = new ProductTableModel(products);
            table.setModel(model);
        });

        //注册"重置"按钮的 ActionEvent 事件监听器
        btnReset.addActionListener(e -> {
            products = dao.findAll();                                   ③
            TableModel model = new ProductTableModel(products);
            table.setModel(model);
        });

        return searchPanel;
    }

    //初始化右侧面板
    private JPanel getRightPanel() {
```

```
JPanel rightPanel = new JPanel();
rightPanel.setBackground(Color.WHITE);

rightPanel.setLayout(new GridLayout(2, 1, 0, 0));

lblImage = new JLabel();
rightPanel.add(lblImage);
lblImage.setHorizontalAlignment(SwingConstants.CENTER);

JPanel detailPanel = new JPanel();
detailPanel.setBackground(Color.WHITE);
rightPanel.add(detailPanel);
detailPanel.setLayout(new GridLayout(8, 1, 0, 5));

JSeparator separator_1 = new JSeparator();
detailPanel.add(separator_1);

lblListprice = new JLabel();
detailPanel.add(lblListprice);
//设置字体
lblListprice.setFont(new Font("微软雅黑", Font.PLAIN, 16));

lblUnitcost = new JLabel();
detailPanel.add(lblUnitcost);
//设置字体
lblUnitcost.setFont(new Font("微软雅黑", Font.PLAIN, 16));

lblDescn = new JLabel();
detailPanel.add(lblDescn);
//设置字体
lblDescn.setFont(new Font("微软雅黑", Font.PLAIN, 16));

JSeparator separator_2 = new JSeparator();
detailPanel.add(separator_2);

JButton btnAdd = new JButton("添加到购物车");
btnAdd.setFont(new Font("微软雅黑", Font.PLAIN, 15));
detailPanel.add(btnAdd);

//布局占位使用
JLabel lb1 = new JLabel("");
detailPanel.add(lb1);

JButton btnCheck = new JButton("查看购物车");
btnCheck.setFont(new Font("微软雅黑", Font.PLAIN, 15));
detailPanel.add(btnCheck);

//注册"添加到购物车"按钮的 ActionEvent 事件监听器
btnAdd.addActionListener(e -> {                                          ④
```

```
            if (selectedRow < 0) {
                return;
            }
            //添加商品到购物车处理
            Product selectProduct = products.get(selectedRow);
            String productid = selectProduct.getProductid();

            if (cart.containsKey(productid)) {           //购物车中已经有该商品    ⑤
                //获得商品数量
                Integer quantity = cart.get(productid);                          ⑥
                cart.put(productid, ++quantity);                                 ⑦
            } else {                                     //购物车中还没有该商品
                cart.put(productid, 1);                                          ⑧
            }

            System.out.println(cart);
        });

        //注册"查看购物车"按钮的 ActionEvent 事件监听器
        btnCheck.addActionListener(e -> {                                        ⑨
            CartFrame cartFrame = new CartFrame(cart, this);
            cartFrame.setVisible(true);
            setVisible(false);
        });

        return rightPanel;
    }

    //初始化左侧面板
    private JScrollPane getLeftPanel() {

        JScrollPane leftScrollPane = new JScrollPane();
        //将表格作为滚动面板的各个视口视图
        leftScrollPane.setViewportView(getTable());
        return leftScrollPane;
    }

    //初始化左侧面板中的表格控件
    private JTable getTable() {

        TableModel model = new ProductTableModel(this.products);

        if (table == null) {
            table = new JTable(model);
            //设置表中内容字体
            table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
            //设置表列标题字体
            table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
            //设置表行高
            table.setRowHeight(51);
            table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

```
            ListSelectionModel rowSelectionModel = table.getSelectionModel();

            rowSelectionModel.addListSelectionListener(e -> {                    ⑩

               //只处理鼠标释放
               if (e.getValueIsAdjusting()) {
                   return;
               }

                ListSelectionModel lsm = (ListSelectionModel) e.getSource();
                selectedRow = lsm.getMinSelectionIndex();
                if (selectedRow < 0) {
                    return;
                }
                //更新右侧面板内容
                Product p = products.get(selectedRow);
                String petImage = String.format("/images/%s",
    p.getImage());                                                               ⑪
                ImageIcon icon = new ImageIcon(ProductListFrame.class
    .getResource(petImage));                                                     ⑫
                lblImage.setIcon(icon);

                String descn = p.getDescn();
                lblDescn.setText("商品描述: " + descn);

                double listprice = p.getListprice();
                String slistprice = String.format("商品市场价: %.2f", listprice);
                lblListprice.setText(slistprice);

                double unitcost = p.getUnitcost();
                String slblUnitcost = String.format("商品单价: %.2f", unitcost);
                lblUnitcost.setText(slblUnitcost);
            });
        } else {
            table.setModel(model);
        }
        return table;
    }

}
```

上述代码第①行的 Products = dao.findAll()语句是查询所有数据,单击“重置”按钮也调用 dao.findAll()语句查询所有数据,见代码第③行。代码第②行的代码块是用户单击“查询”按钮调用的。

代码第④行的代码块是用户单击“添加到购物车”按钮调用的,其中代码第⑤行判断购物车中是否已经有了选中的商品,如果有,则通过代码第⑥行取出商品数量,代码第⑦行是将商品数量加 1 后,再重新放回到购物车中;如果没有商品,则将该商品添加到购物车中,商品数量为 1。

代码第⑨行是单击“查看购物车”按钮时调用的代码块。此时当前界面会调转到购物车窗口。

代码第⑩行是用户选中表格中某一行时调用的代码块，在这里根据用户选中的商品更新右边的详细商品信息。

代码第⑪行是获得图片相对路径，它们属于资源目录(即 src 源文件夹)。代码第⑫行的 ProductListFrame.class.getResource(petImage)语句可以获得图片文件运行时的绝对路径。

商品列表窗口中使用了自定义表格模型 ProductTableModel。ProductTableModel 代码如下：

```
//ProductTableModel.java 文件
package com.a51work6.jpetstore.ui;
…
//商品列表表格模型
public class ProductTableModel extends AbstractTableModel {

    //表格列名 columnNames
    private String[] columnNames = { "商品编号", "商品类别", "商品中文名",
"商品英文名" };

    //表格中的内容保存在 List<Product>集合中
    private List<Product> data = null;

    public ProductTableModel(List<Product> data) {
        this.datas = data;
    }

    //返回列数
    @Override
    public int getColumnCount() {
        return columnNames.length;
    }

    //返回行数
    @Override
    public int getRowCount() {
        return data.size();
    }

    //获得某行某列的数据,而数据保存在对象数组 data 中
    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {

        //每一行就是一个 Product 商品对象
        Product p = data.get(rowIndex);

        switch(columnIndex) {
        case 0:
```

```
                return p.getProductid();                //第一列商品编号
            case 1:
                return p.getCategory();                 //第二列商品类别
            case 2:
                return p.getCname();                    //第三列商品中文名
            default:
                return p.getEname();                    //第四列商品英文名
            }
        }

        @Override
        public String getColumnName(int columnIndex) {
            return columnNames[columnIndex];
        }
    }
```

上述表格模型代码继承了 AbstractTableModel 抽象类，表格中的数据保存在 List < Product >集合中。类似的表格模型在 23.6 节介绍过，这里不再赘述。

B.5.5 迭代 4.5：商品购物车窗口

当用户在商品列表窗口单击“查看购物车”按钮，则会跳转到商品购物车窗口，如图 B-16 所示。在该窗口可进行的操作如下。



图 B-16　商品购物车窗口

- 返回商品列表。当用户单击“返回商品列表”按钮时，界面跳转回上一级窗口（商品列表窗口），用户还可以重新添加新的商品到购物车。

- 修改商品数量。用户如果想修改商品数量，可以在购物车表格中双击某一商品数量单元格，使其进入编辑状态。用户只能输入大于或等于 0 的数值，不能输入负数或非数值字符。
- 提交订单。如果商品选择完成，用户想提交订单，则可以单击“提交订单”按钮生成订单，订单生成后会在数据库中插入订单信息和订单明细信息。然后会弹出如图 B-17 所示订单等待付款确认对话框。如果单击“是”按钮，则进入付款流程，由于付款需要实际的支付接口，因此付款功能未实现。如果单击“否”按钮，则退出系统。

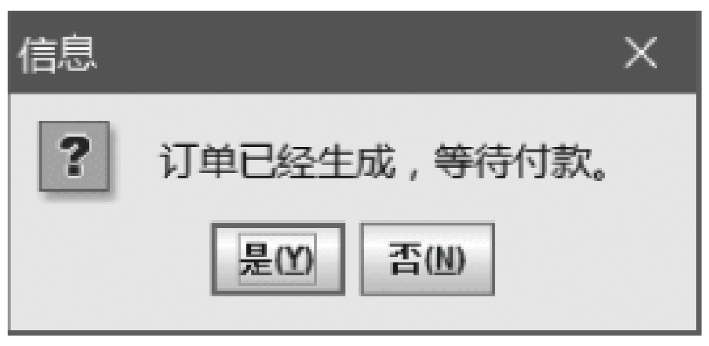


图 B-17　订单生成确认对话框

商品购物车窗口 CartFrame 代码如下：

```
//CartFrame.java 文件
package com.a51work6.jpetstore.ui;
…
//商品购物车窗口
public class CartFrame extends MyFrame {

    private JTable table;

    //购物车数据
    private Object[][] data = null;

    //创建商品 Dao 对象
    private ProductDao dao = new ProductDaoImp();

    //购物车,键是选择的商品 Id,值是商品的数量
    private Map<String, Integer> cart;
    //引用到上级 Frame(ProductListFrame)
    private ProductListFrame productListFrame;

    public CartFrame(Map<String, Integer> cart, ProductListFrame product ListFrame) {

        super("商品购物车", 1000, 700);
        this.cart = cart;
        this.productListFrame = productListFrame;

        JPanel topPanel = new JPanel();
        FlowLayout fl_topPanel = (FlowLayout) topPanel.getLayout();
        fl_topPanel.setVgap(10);
        fl_topPanel.setHgap(20);
        getContentPane().add(topPanel, BorderLayout.NORTH);

        JButton btnReturn = new JButton("返回商品列表");
        btnReturn.setFont(new Font("微软雅黑", Font.PLAIN, 15));
        topPanel.add(btnReturn);
```

```
        JButton btuSubmit = new JButton("提交订单");
        topPanel.add(btuSubmit);
        btuSubmit.setFont(new Font("微软雅黑", Font.PLAIN, 15));

        JScrollPane scrollPane = new JScrollPane();
        getContentPane().add(scrollPane, BorderLayout.CENTER);
        scrollPane.setViewportView(getTable());

        //注册"提交订单"按钮的 ActionEvent 事件监听器
        btuSubmit.addActionListener(e -> {                                        ①
            //生成订单
            generateOrders();

            JLabel label = new JLabel("订单已经生成,等待付款.");
            label.setFont(new Font("微软雅黑", Font.PLAIN, 15));
            if (JOptionPane.showConfirmDialog(this, label, "信息",
                        JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) {
                //TODO 付款
                System.exit(0);
            } else {
                System.exit(0);
            }
        });

        //注册"返回商品列表"按钮的 ActionEvent 事件监听器
        btnReturn.addActionListener(e -> {
            //更新购物车
            for (int i = 0; i < data.length; i++) {
                //商品编号
                String productid = (String) data[i][0];
                //数量
                Integer quantity = (Integer) data[i][3];
                cart.put(productid, quantity);
            }
            this.productListFrame.setVisible(true);
            setVisible(false);
        });
    }

    //初始化左侧面板中的表格控件
    private JTable getTable() {

        //准备表中数据
        data = new Object[cart.size()][5];
        Set<String> keys = this.cart.keySet();
        int indx = 0;
        for (String productid : keys) {
            Product p = dao.findById(productid);
            data[indx][0] = p.getProductid();                    //商品编号
            data[indx][1] = p.getCname();                        //商品名
```

```
            data[indx][2] = new Double(p.getUnitcost());              //商品单价
            data[indx][3] = new Integer(cart.get(productid));         //数量
            //计算商品应付金额
            double amount = (double) data[indx][2] * (int) data[indx][3];
            data[indx][4] = new Double(amount);
            indx++;
        }

        //创建表数据模型
        TableModel model = new CartTableModel(data);

        if (table == null) {
            //创建表
            table = new JTable(model);
            //设置表中内容字体
            table.setFont(new Font("微软雅黑", Font.PLAIN, 16));
            //设置表列标题字体
            table.getTableHeader().setFont(new Font("微软雅黑", Font.BOLD, 16));
            //设置表行高
            table.setRowHeight(51);
            table.setRowSelectionAllowed(false);

        } else {
            table.setModel(model);
        }
        return table;
    }

    //生成订单
    private void generateOrders() {                                         ②

        OrderDao orderDao = new OrderDaoImp();
        OrderDetailDao orderDetailDao = new OrderDetailDaoImp();

        Order order = new Order();
        order.setUserid(MainApp.accout.getUserid());                        ③
        //0 待付款
        order.setStatus(0);
        //订单 Id 是当前时间
        Date now = new Date();
        long orderId = now.getTime();
        order.setOrderid(orderId);                                          ④
        order.setOrderdate(now);
        order.setAmount(getOrderTotalAmount());                             ⑤

        //下订单时间是数据库自动生成不用设置
        //创建订单
        orderDao.create(order);                                             ⑥

        for (int i = 0; i < data.length; i++) {
```

```
                OrderDetail orderDetail = new OrderDetail();
                orderDetail.setOrderid(orderId);
                orderDetail.setProductid((String) data[i][0]);
                orderDetail.setQuantity((int) data[i][3]);
                orderDetail.setUnitcost((double) data[i][2]);
                //创建订单详细
                orderDetailDao.create(orderDetail);                              ⑦
            }
        }

        //计算订单应付总金额
        private double getOrderTotalAmount() {

            double totalAmount = 0.0;
            for (int i = 0; i < data.length; i++) {
                //计算商品应付金额
                totalAmount += (Double) data[i][4];
            }
            return totalAmount;
        }
    }
```

当用户单击“提交订单”按钮时调用代码第①行的代码块。在该代码块中首先调用 generateOrders()方法生成订单，然后通过调用 JOptionPane.showConfirmDialog 方法弹出付款确认对话框。

代码第②行是生成订单 generateOrders()方法定义，在该方法中将订单信息插入到数据库订单表和订单明细表中。代码第③行设置订单中用户 Id 属性，这个属性是在登录时保存在 MainApp.accout 静态变量中的。代码第④行设置订单 Id 属性，订单 Id 生成规则是当前系统时间毫秒数，这种生成规则在用户访问量少的情况下可以满足要求。代码第⑤行设置该订单应付金额，该金额的计算是通过 getOrderTotalAmount()方法实现的，就是将订单中所有商品价格乘以数量，然后累加起来。

代码第⑥行是将订单数据插入到数据库中，由于订单中有可能有多个商品，所以代码第⑦行循环插入订单明细数据。

订单生成后可以在数据中查看生成的结果，如图 B-18 所示。

购物车窗口中会用到购物车表格，购物车表格比较复杂，用户可以修改数量这一列，其他的列不能修改。另外，修改的数量是要验证的，不能小于 0，更不能输入非数值字符。这些需求的解决是通过自定义表格模型实现的。表格模型 CartTableModel 代码如下：

```
//CartTableModel.java 文件
package com.a51work6.jpetstore.ui;

import javax.swing.table.AbstractTableModel;

//购物车表格模型
public class CartTableModel extends AbstractTableModel {
```

```
    //表格列名 columnNames
    private String[] columnNames = { "商品编号", "商品名", "商品单价", "数量", "商品应付金
额" };

    //表格中数据保存在 data 二维数组中
    private Object[][] data = null;

    public CartTableModel(Object[][] data) {
        this.data = data;
    }

    //返回列数
    @Override
    public int getColumnCount() {
        return columnNames.length;
    }

    //返回行数
    @Override
    public int getRowCount() {
        return data.length;
    }

    //获得某行某列的数据,而数据保存在对象数组 data 中
    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        return data[rowIndex][columnIndex];
    }

    @Override
    public String getColumnName(int columnIndex) {
        return columnNames[columnIndex];
    }

    @Override
    public boolean isCellEditable(int rowIndex, int columnIndex) {          ①
        //数量列可以修改
        if (columnIndex == 3) {
            return true;
        }
        return false;
    }

    @Override
    public void setValueAt(Object aValue, int rowIndex, int columnIndex) {  ②
        //只允许修改数量列
        if (columnIndex != 3) {                                             ③
            return;
        }
        try {
```

```
                //从表中获得修改之后的商品数量,表中的数据都是 String 类型
                int quantity = new Integer((String) aValue);                  ④
                //商品数量不能小于 0
                if (quantity < 0) {                                           ⑤
                    return;
                }
                //更新数量列
                data[rowIndex][3] = quantity;                                 ⑥
                //计算商品应付金额
                double unitcost = (double) data[rowIndex][2];                 ⑦
                double totalPrice = unitcost * quantity;                      ⑧
                //更新商品应付金额列
                data[rowIndex][4] = new Double(totalPrice);                   ⑨

            } catch (Exception e) {
            }
        }
    }
```

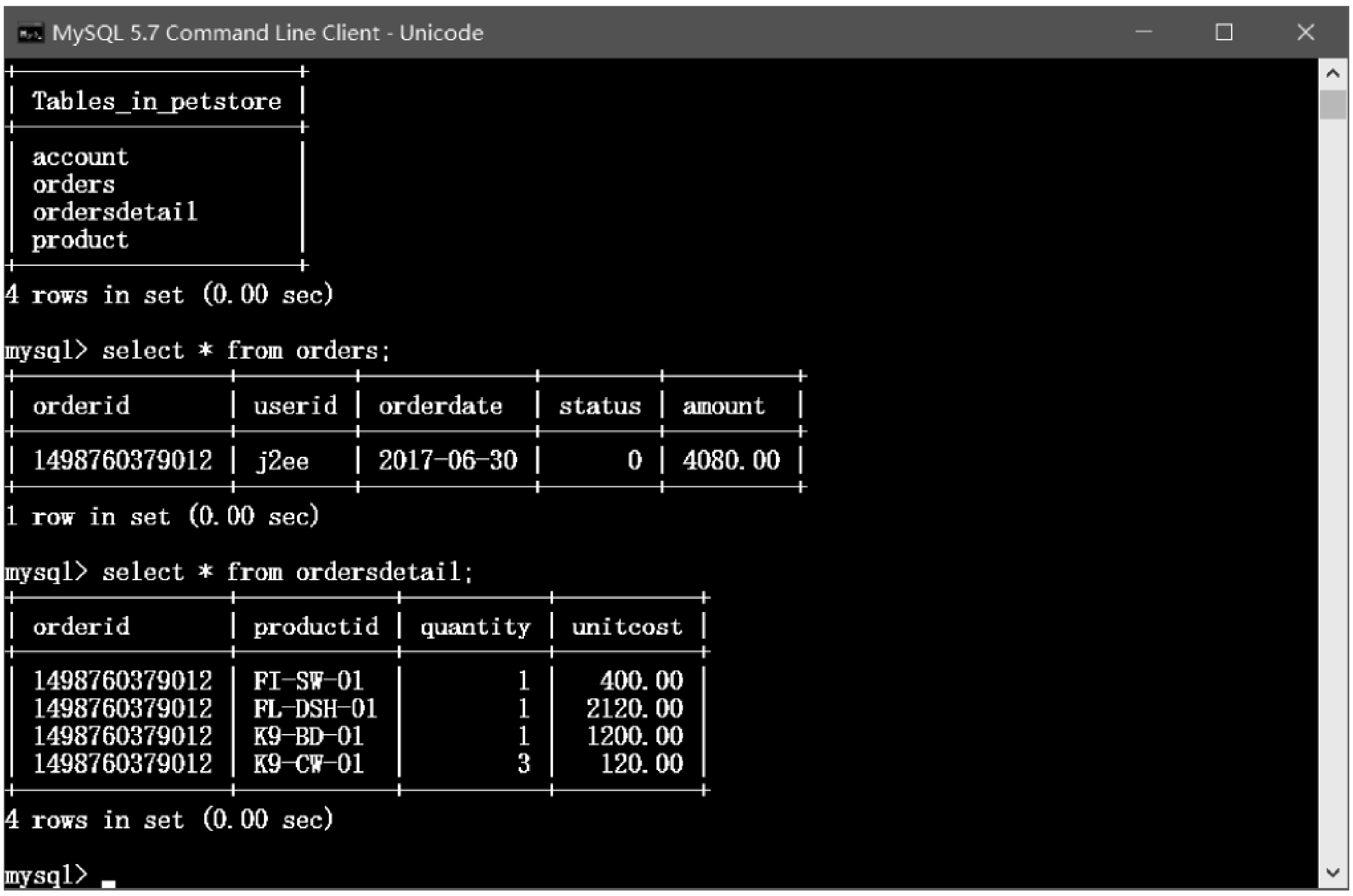


图 B-18　订单生成数据

为了能让表格可以被编辑，需要覆盖代码第①行的 isCellEditable 方法，在该方法中判断当前列索引是 3（就是数量列），则返回 true，表示这一列可以修改。

在修改数量时需要进行验证，则需要覆盖代码第②行的 setValueAt 方法，其中 aValue 参数是当前单元格（rowIndex，columnIndex）的输入值。代码第③行判断数量列才进行处理。代码第④行将输入值 aValue 转换为整数，如果是非数值字符，则会发生异常，结束 setValueAt 方法。代码第⑤行判断小于 0 时结束 setValueAt 方法。代码第⑥行用输入值

aValue 替换二维数组 data 中对应的数据。代码第⑦行 data[rowIndex][2]取出二维数组商品单价。代码第⑧行计算商品应付金额，然后通过代码第⑨行将商品应付金额更新为二维数组 data 中的商品应付金额元素。

B.6 任务 5：应用程序打包发布

编写的 Java 程序最后要被人使用，大多数人不会用 JDK 指令或 Eclipse 工具运行 java 程序，而且一个 java 项目可能有很多字节码文件，使用起来也不好管理。因此，最后发布时需要给应用程序打包。

B.6.1 迭代 5.1：处理 TODO、FIXME 和 XXX 任务

在最后发布打包之前，还需要处理一些任务，其中首先应该处理代码中的 TODO、FIXME 和 XXX 注释任务。这三种注释称为“地标注释”。这种注释虽然不是 Java 官方所提供的，但是主流语言和主流的 IDE 工具也都支持“地标注释”。

Eclipse 工具支持如下三种地标注释。

- TODO：说明此处有待处理的任务，或代码没有编写完成。
- FIXME：说明此处代码是错误的，需要修正。
- XXX：说明此处代码虽然实现了功能，但是实现的方法有待商榷，希望将来能改进。

PetStore 项目待处理任务如图 B-19 所示。

Problems Javadoc 声明 控制台 Diagrams 任务

14 项

!	描述	资源	路径	位置	类型
	TODO 付款	CartFrame.java	/PetStore/src/com/a51work6/jpetstore/ui	第 78 行	Java 任务
	TODO 自动生成的方法存根	AccountDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 19 行	Java 任务
	TODO 自动生成的方法存根	AccountDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 86 行	Java 任务
	TODO 自动生成的方法存根	AccountDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 92 行	Java 任务
	TODO 自动生成的方法存根	AccountDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 98 行	Java 任务
	TODO 自动生成的方法存根	OrderDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 81 行	Java 任务
	TODO 自动生成的方法存根	OrderDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 87 行	Java 任务
	TODO 自动生成的方法存根	OrderDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 93 行	Java 任务
	TODO 自动生成的方法存根	OrderDetailDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 20 行	Java 任务
	TODO 自动生成的方法存根	OrderDetailDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 110 行	Java 任务
	TODO 自动生成的方法存根	OrderDetailDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 116 行	Java 任务
	TODO 自动生成的方法存根	ProductDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 179 行	Java 任务
	TODO 自动生成的方法存根	ProductDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 185 行	Java 任务
	TODO 自动生成的方法存根	ProductDaoImp.java	/PetStore/src/com/a51work6/jpetstore/dao/mysql	第 191 行	Java 任务

图 B-19 处理 TODO、FIXME 和 XXX 任务

请根据需要处理这些任务。

B.6.2 迭代 5.2：处理警告

Java 的程序代码往往还有很多警告，这些警告也不可忽视，应该检查一下那些是程序本身的问题，然后加以修正。这些警告可以在图 B-20 所示问题视图中查看到。

如果问题视图没有打开，可以通过选择“窗口”→“显示视图”→“问题”命令，打开如图 B-20 所示的视图，双击其中的问题可以跳转到代码处。



图 B-20　处理警告任务

B.6.3　迭代 5.3：打包

任务和问题都已经检查并修正后，即可打包。在 JDK 中有一个 jar 命令，它可以为 Java 字节码文件进行打包，打包之后的文件一般是.jar 文件，该文件是 zip 压缩格式。使用 jar 文件有很多好处，首先，文件经过压缩占用空间小，其次，文件多个字节码、资源和配置文件被打包成一个文件方便管理。

要发布的 Java 项目的类型不同，打包的内容也会有所不同。主要是注意 Java 应用程序(Java application)①的项目打包，与其他的 Java 项目有所不同。Java 应用程序在打包时需要指定包含 main 主方法的类是哪一个。PetStore 项目属于 Java 应用程序项目，下面来介绍一下 PetStore 项目打包过程。

可以使用 JDK 中的 jar 命令或者使用 Eclipse 等 IDE 工具进行打包。

1. 使用 jar 命令打包

jar 命令模拟 UNIX 和 Linux 中的 tar 命令，参数也类似。常用的如下。

- -c：创建新文档。
- -x：从文档中解压文件。
- -v：输出压缩或解压信息。
- -f：指定文档文件名。
- -m：指定一个自定义清单文件。
- -C：指定打包的目录，默认是当前目录，需要打包其他目录下的文件时，则需要使用此参数。

首先需要创建一个打包目录，将需要打包的文件复制到这里，如图 B-21 所示，将 Eclipse 编译之后的 bin 目录复制为打包目录，lib 目录中放置第三方类库文件，PetStore 项目需要 MySQL 的 JDBC 驱动程序 mysql-connector-java-5.1.41-bin.jar，该文件需要复制到 lib 目录。打包目录下还有一

```
<PetStore打包目录>
├─ bin
│   ├─ com
│   │   └─ a51work6
│   │       └─ jpetstore
│   │           ├─ dao
│   │           │   ├─ AccountDao.class
│   │           │   ...
│   │           │   └─ ProductDao.class
│   │           ├─ domain
│   │           │   ├─ Account.class
│   │           │   ...
│   │           │   └─ Product.class
│   │           └─ ui
│   │               ├─ CartFrame.class
│   │               ...
│   │               └─ ProductTableModel.class
│   └─ images
│       ├─ bird1.gif
│       ...
│       ├─ snake1.jpg
│       └─ splash.gif
├─ lib
│   └─ mysql-connector-java-5.1.41-bin.jar
└─ mymanifest.txt
```

图 B-21　打包目录结构

① Java 应用程序是指包含有 main 主方法的类，通过该类能够访问其中的 Java 程序。

个 mymanifest. txt 文件，它自己编写的清单（Manifest）文件，所有的 jar 文件都包含一个清单文件，如果不提供清单文件，则 jar 命令会生成该文件。mymanifest. txt 文件内容如下：

```
Manifest-Version: 1.0
Class-Path: . ./lib/mysql-connector-java-5.1.41-bin.jar
Main-Class: com.a51work6.jpetstore.ui.MainApp
```

其中，Manifest-Version 指定清单文件版本号；Class-Path 指定类路径，多个类路径之间用空格分隔，mymanifest. txt 中有两个类路径，“.”表示当前路径，./lib/mysql-connector-java-5.1.41-bin.jar 表示当前 lib 目录下的 mysql-connector-java-5.1.41-bin.jar 文件；Main-Class 指定主类文件，就是包含 main 主方法的类。

启动命令行工具，进入到 PetStore 目录，输入如下指令：

```
jar -cvfm PetStore.jar mymanifest.txt -C bin . ./lib
```

其中，-cvfm 参数是 c、v、f、m 四个参数的叠加，PetStore. jar 打包生成的文件 mymanifest. txt 指定的清单文件；-C 参数指定默认目录是 bin 目录；“.”代表 bin 目录所有文件都打包到 PetStore. jar 中；./lib 表示把该目录也打包到 PetStore. jar 中。

输入指令后按 Enter 键，终端窗口会输出打包信息。如果成功，则会在当前目录下生成一个 PetStore. jar 文件。PetStore. jar 文件可以使用 jar 解压，由于 PetStore. jar 是 Java 应用程序，所以可以运行，最简单的运行方式是双击该文件即可运行。也可以使用 Java 命令，在命令行中输入如下命令：

```
java -jar PetStore.jar
```

2. 使用 Eclipse IDE 工具进行打包

使用 Eclipse IDE 工具进行打包非常简单，在 Eclipse 中选择 PetStore 项目，右击，从弹出的快捷菜单中选择“导出”命令，弹出如图 B-22 所示对话框，“可运行的 JAR 文件”选项是打包 Java 应用程序时使用，普通程序打包选择“JAR 文件”。

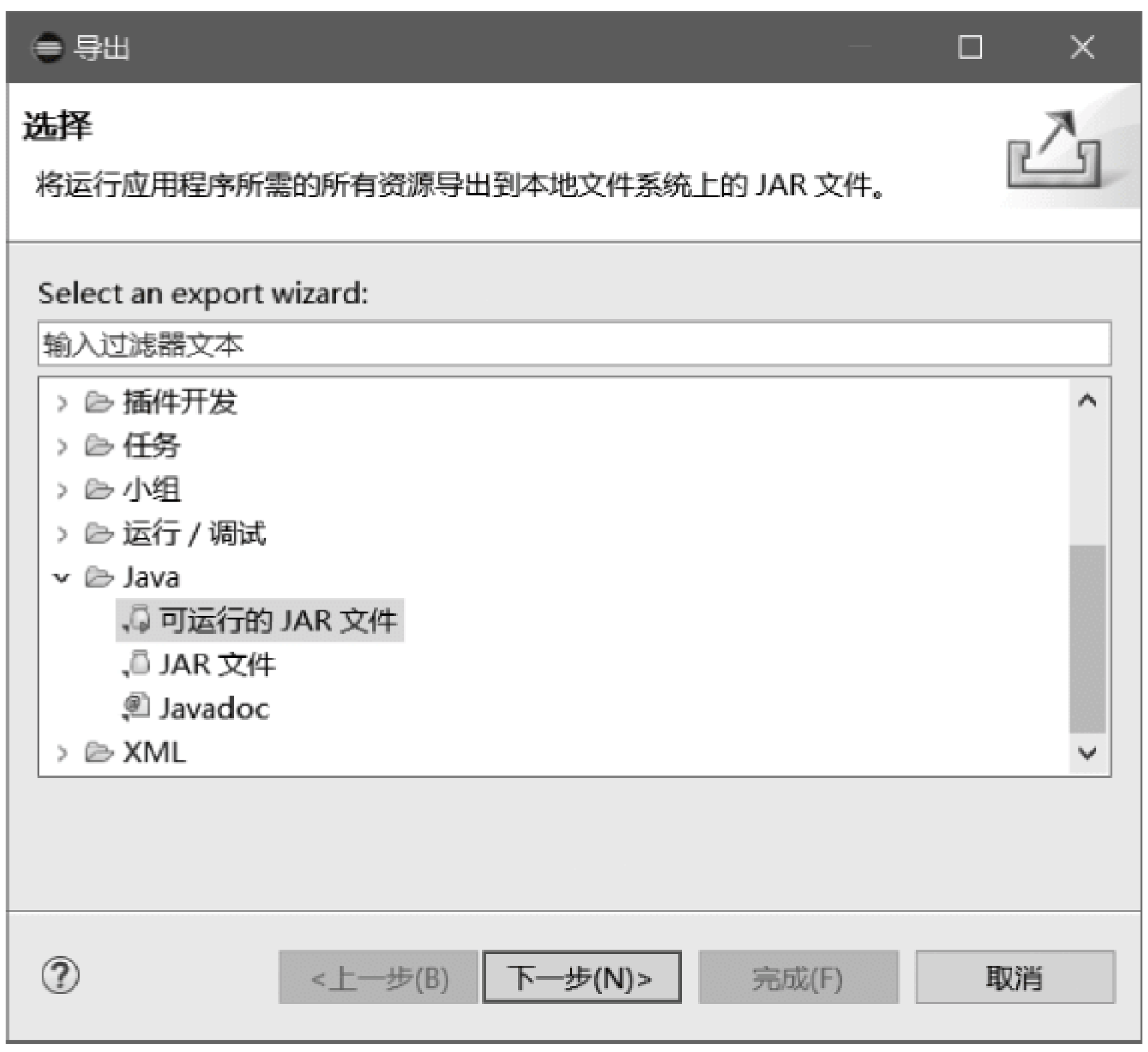


图 B-22 导出文件对话框

在图 B-22 中选择“可运行的 JAR 文件”后，单击“下一步”按钮，进入如图 B-23 所示对话框。这里可以选择第三方库的处理方式。



图 B-23　第三方库处理方式

(1) 处理方式是将库中所需要的字节码文件提取出来，打包到 jar 文件中，这种方式是推荐的做法。

(2) 处理方式是将库直接打包到 jar 文件中，然后在清单文件中指定类路径，类似于前文介绍的 jar 打包方式。

(3) 处理方式是将库不打包到 jar 文件中，而放置在与 jar 文件所在目录下的 XXX_lib 目录中。这种方式发布的时候需要把 jar 文件和 XXX_lib 目录一同发布。

打包文件生成之后就可以双击运行了。

附录 C APPENDIX C 同步练习参考答案

第 1 章

1. 答案：参考 1.2 节
2. 答案：参考 1.3 节

第 2 章

1. 答案：参考 2.1 节
2. 答案：参考 2.2 节

第 3 章

1. 答案：参考 3.1 节
2. 答案：B

第 4 章

1. 答案：C
2. 答案：BCDE
3. 答案：(1)错　(2)错

第 5 章

1. 答案：D
2. 答案：A
3. 答案：C
4. 答案：BC

第 6 章

1. 答案：B
2. 答案：C
3. 答案：A
4. 答案：BC

第 7 章

1. 参考答案：

（1）请使用 for 循环计算水仙花数。

```
public class HelloWorld {

    public static void main(String[] args) {
        int x, y, z;

        for (int i = 100; i < 1000; i++) {
            x = i % 10;
            y = i / 100;
            z = (i - y * 100) / 10;
            if (i == (x * x * x + y * y * y + z * z * z)) {
                System.out.println(i);
            }
        }

    }
}
```

输出结果如下：

```
153
370
371
407
```

（2）请使用 while 循环计算水仙花数。

```
public class HelloWorld {

    public static void main(String[] args) {
        int x, y, z;
        int i = 100;

        while ( i < 1000 ) {
            x = i % 10;
            y = i / 100;
            z = (i - y * 100) / 10;
```

```
                if (i == (x * x * x + y * y * y + z * z * z)) {
                    System.out.println(i);
                }
                i++;
            }

        }
    }
```

输出结果如下：

```
153
370
371
407
```

2. 参考答案：

```
public class HelloWorld {

    public static void main(String[] args) {

        for (int i = 1; i < 5; i++) {
            int j = 1;
            while (j <= 4 - i) {
                System.out.print(" ");
                j++;
            }
            j = 1;
            while (j <= 2 * i - 1) {
                System.out.print("*");
                j++;
            }
            System.out.println("");
        }

    }
}
```

3. 答案：B

4. 答案：D

5. 答案：ADE

6. 答案：CD

第 8 章

1. 答案：BD

2. 答案：D

3. 答案：C
4. 答案：B
5. 答案：BC

第 9 章

1. 答案：baseball
2. 答案：B
3. 答案：Java Applet
4. 答案：false，true

第 10 章

1. 答案：D
2. 答案：CD
3. 答案：对
4. 答案：ACD
5. 答案：AD
6. 答案：BCD

第 11 章

1. 答案：C
2. 答案：(1) 对 (2) 错

第 12 章

1. 答案：AD
2. 答案：C
3. 答案：D
4. 答案：D
5. 答案：对
6. 答案：AB

第 13 章

1. 答案：A
2. 答案：(1) 对 (2) 对
3. 答案：C

第 14 章

1．答案：BCD
2．答案：ABC
3．答案：(1) 对　(2) 错

第 15 章

1．答案：ABCD
2．答案：(1) 对　(2) 对
3．答案：AD

第 16 章

1．答案：(1) 对　(2) 错
2．答案：C
3．答案：ABC

第 17 章

1．答案：BCD
2．答案：C
3．答案：C
4．答案：B
5．答案：B
6．答案：参考 17.2.2 节

第 18 章

1．答案：(1) 对　(2) 对　(3) 对　(4) 对　(5) 错
2.答案：B

第 19 章

1．答案：AD
2．答案：(1) 对　(2) 对

第 20 章

1. 答案：AC
2. 答案：C
3. 答案：BE
4. 答案：CE

第 21 章

1. 答案：AD
2. 答案：A
3. 答案：D
4. 答案：B

第 22 章

1. 答案：AC
2. 答案：B
3. 答案：ABC
4. 答案：AB

第 23 章

1. 答案：CD
2. 答案：ABC
3. 答案：C
4. 答案：A